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

Jipeng Zhang[1] Jiaheng Zhang[1]

[1]National University of Singapore, Singapore
jp-zhang@outlook.com

Paper: `https://eprint.iacr.org/2025/1867`
Artifact: `https://github.com/Ji-Peng/VecFalcon`
Slides: `https://ji-peng.github.io/uploads/tches2026/VecFalcon_slides.pdf`
IACR TCHES 2026-1

2026-01-08

Outline

Contributions

This paper focuses on optimizing **Falcon Signature Generation**.

- **Performance Profiling:** Identified `BaseSampler` ($> 30\%$) and FFT-related subroutines (on RISC-V) as bottlenecks.
- **Vectorized BaseSampler:**
  - Implemented across **6 ISAs**: SSE2, AVX2, AVX-512F, NEON, RVV, RV64IM.
  - Achieved up to **8.4**× (AVX2) and **7.7**× (RVV) speedup for the sampler.
- **Vectorized FFT on RVV:**
  - Novel **4+5 layer merging** strategy using strided load/store instructions.
  - Achieved **4.7**× speedup on RVV.
- Signature generation speedups of **23%** (AVX2), **36%** (AVX-512F), and **59%** (RV64GCVB).

Motivations

FALCON (FN-DSA)

- Selected by NIST for standardization (FIPS 206). **Note:** The FIPS 206 standard document was **not yet published** at the time of this work.
- Fast verification, but **slow signature generation**.

Bottleneck 1: Discrete Gaussian Sampling

- `BaseSampler` accounts for $> 30\%$ of signing time.
- Non-vectorizable in reference code due to sequential `UniformBits` calls for KAT compatibility.

Bottleneck 2: FFT on RISC-V

- FFT-related ops take $\approx 38\%$ time on SpacemiT X60 (RV64GCVB).
- Existing optimizations lack efficient deep layer merging strategies.

Background: Falcon & BaseSampler

Falcon Signature Generation

- Involves Fast Fourier Sampling (`ffSampling`) and Discrete Gaussian Sampling (`SamplerZ`).
- `SamplerZ` calls `BaseSampler` to sample integers z_0 from distribution χ.

Target Platforms

Architecture	CPU	ISA
x86-64	Intel i7-11700K	SSE2, AVX2, AVX-512F
ARMv8-A	Cortex-A72	NEON
RISC-V	SpacemiT X60	RV64GC, RVV (v1.0), Bit-manip

Background: Falcon & BaseSampler

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

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

Output : A signature sig of message m

```
1:  $\mathsf{r} \leftarrow \{0,1\}^{320}$ uniformly
2:  $c \leftarrow \mathsf{HashToPoint}(\mathsf{r}\|\mathsf{m}, q, n)$
3:  $\mathbf{t} \leftarrow (-\frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(f))$
4:  do
5:      do
6:          $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$
7:          $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
8:      while $\|\mathbf{s}\|^2 > \lfloor\beta^2\rfloor$
9:      $(s_1, s_2) \leftarrow \mathsf{iFFT}(\mathbf{s})$
10:     $\mathsf{s} \leftarrow \mathsf{Compress}(s_2, 8 \cdot \mathsf{sbytelen} - 328)$
11: while $\mathsf{s} = \perp$
12: 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 : $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathbb{Z}[x]/(x^n+1))^2$

```
1:  if $n = 1$ then
2:      $\sigma' \leftarrow \mathsf{T.value}$
3:      $z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma')$
4:      $z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma')$
5:      return $\mathbf{z} = (z_0, z_1)$
6:  $(\ell, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
7:  $\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$
8:  $\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$
9:  $z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$
10: $t_0' \leftarrow t_0 + (t_1 - z_1) \odot \ell$
11: $\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t_0')$
12: $\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$
13: $z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
14: return $\mathbf{z} = (z_0, z_1)$
```

Background: Falcon & BaseSampler

Algorithm 3: SamplerZ(μ, σ')

```
Input   : $\mu, \sigma' \in \mathcal{R}$; $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
Output  : $z \in \mathbb{Z}$ close to $D_{\mathbb{Z},\mu,\sigma'}$
1: $r \leftarrow \mu - \lfloor \mu \rfloor$
2: $ccs \leftarrow \sigma_{\min}/\sigma'$
3: while (1) do
4:     $z_0 \leftarrow$ BaseSampler()
5:     $b \leftarrow$ UniformBits(8) & 0x1
6:     $z \leftarrow b + (2 \cdot b - 1) z_0$
7:     $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2}$
8:     if BerExp($x$, $ccs$) $= 1$ then
9:         return $z + \lfloor \mu \rfloor$
```

Algorithm 4: ApproxExp(x, ccs)

```
Input   : $x \in [0, \ln(2)]$; $ccs \in [0, 1]$;
          precomputed array $C$
Output  : An integer
          $\approx 2^{63} \cdot ccs \cdot \exp(-x)$
1: $y \leftarrow C[0]$
2: $z \leftarrow \lfloor 2^{63} \cdot x \rfloor$
3: for $i = 1$ to 12 do
4:     $y \leftarrow C[i] - (z \cdot y) \gg 63$
5: $z \leftarrow \lfloor 2^{63} \cdot ccs \rfloor$
6: $y \leftarrow (z \cdot y) \gg 63$
7: return $y$
```

Algorithm 5: BaseSampler()

```
Output : $z_0 \in \{0, \ldots, 18\}$; $z_0 \sim \chi$
1: $u \leftarrow$ UniformBits(72)
2: $z_0 \leftarrow 0$
3: for $i = 0$ to 17 do
4:     $z_0 \leftarrow z_0 + [\![u < \mathsf{RCDT}[i]]\!]$
5: 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
          $\approx ccs \cdot \exp(-x)$
1: $s \leftarrow \lfloor x / \ln(2) \rfloor$
2: $r \leftarrow x - s \cdot \ln(2)$
3: $s \leftarrow \min(s, 63)$
4: $z \leftarrow (2 \cdot$ ApproxExp($r$, $ccs$) $- 1) \gg s$
5: $i \leftarrow 64$
6: do
7:     $i \leftarrow i - 8$
8:     $w \leftarrow$ UniformBits(8) $- ((z \gg$
        $i)$ & 0xFF)
9: while $((w = 0)$ and $(i > 0))$
10: return $[\![w < 0]\!]$
```

Performance Profiling

Profiling Methodology

- **Baseline:** `C-FN-DSA` project (SHAKE256X4 variant). FALCON-1024's signature generation.
- **Tool:** `gperftools`.
- **Platforms:** Intel i7-11700K (AVX2) & SpacemiT X60 (RV64GCVB).

Key Observations

- **BaseSampler** is a consistent bottleneck ($> 30\%$) across architectures.
- **FFT-related** operations are expensive specifically on RISC-V ($\approx 38\%$).
- *Note: SHA-3 is already optimized for AVX2; BerExp is left for future work.*

Table: Breakdown of Execution Time

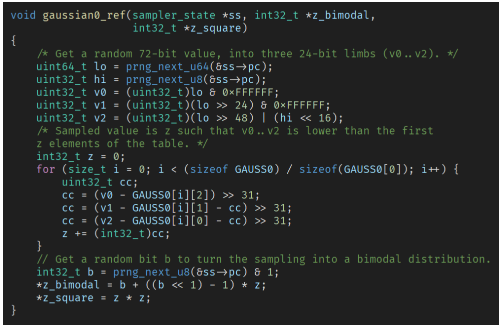
Component	AVX2	RISC-V
BaseSampler	30.2%	30.3%
FFT-related	15.1%	37.6%
SHA-3	22.6%	20.6%
BerExp	31.2%	14.3%

Optimization Targets

Based on this data, we focus on:

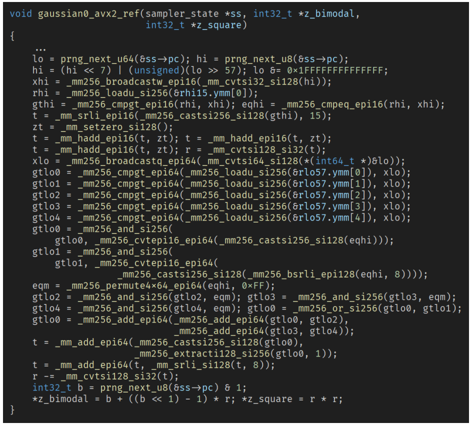
1. **BaseSampler**
2. **FFT-related ops** (RISC-V only)

Vectorized BaseSampler: Previous Methods on x86



```c
void gaussian0_ref(sampler_state *ss, int32_t *z_bimodal,
                   int32_t *z_square)
{
    /* Get a random 72-bit value, into three 24-bit limbs (v0..v2). */
    uint64_t lo = prng_next_u64(&ss->pc);
    uint32_t hi = prng_next_u8(&ss->pc);
    uint32_t v0 = (uint32_t)lo & 0xFFFFFF;
    uint32_t v1 = (uint32_t)(lo >> 24) & 0xFFFFFF;
    uint32_t v2 = (uint32_t)(lo >> 48) | (hi << 16);
    /* Sampled value is z such that v0..v2 is lower than the first
    z elements of the table. */
    int32_t z = 0;
    for (size_t i = 0; i < (sizeof GAUSS0) / sizeof(GAUSS0[0]); i++) {
        uint32_t cc;
        cc = (v0 - GAUSS0[i][2]) >> 31;
        cc = (v1 - GAUSS0[i][1] - cc) >> 31;
        cc = (v2 - GAUSS0[i][0] - cc) >> 31;
        z += (int32_t)cc;
    }
    // Get a random bit b to turn the sampling into a bimodal distribution.
    int32_t b = prng_next_u8(&ss->pc) & 1;
    *z_bimodal = b + ((b << 1) - 1) * z;
    *z_square = z * z;
}
```

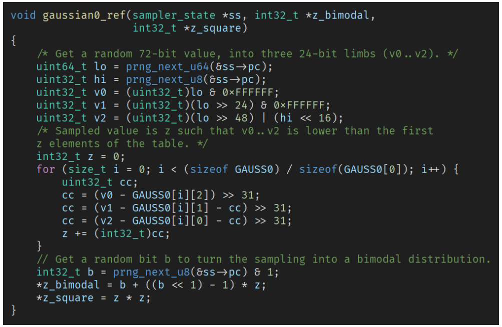
C impl. in `C-FN-DSA` project (72 bits = 3×24 bits; 59 cycles without PRNG overhead)



```c
void gaussian0_avx2_ref(sampler_state *ss, int32_t *z_bimodal,
                        int32_t *z_square)
{
    ...
    lo = prng_next_u64(&ss->pc); hi = prng_next_u8(&ss->pc);
    hi = (hi << 7) | (unsigned)(lo >> 57); lo &= 0x1FFFFFFFFFFFFFF;
    xhi = _mm256_broadcastw_epi16(_mm_cvtsi32_si128(hi));
    rhi = _mm256_loadu_si256(&rhi15.ymm[0]);
    gthi = _mm256_cmpgt_epi16(rhi, xhi); eqhi = _mm256_cmpeq_epi16(rhi, xhi);
    t = _mm_srli_epi16(_mm256_castsi256_si128(gthi), 15);
    zt = _mm_setzero_si128();
    t = _mm_hadd_epi16(t, zt); t = _mm_hadd_epi16(t, zt);
    t = _mm_hadd_epi16(t, zt); r = _mm_cvtsi128_si32(t);
    xlo = _mm256_broadcastq_epi64(_mm_cvtsi64_si128(*(int64_t *)&lo));
    gtlo0 = _mm256_cmpgt_epi64(_mm256_loadu_si256(&rlo57.ymm[0]), xlo);
    gtlo1 = _mm256_cmpgt_epi64(_mm256_loadu_si256(&rlo57.ymm[1]), xlo);
    gtlo2 = _mm256_cmpgt_epi64(_mm256_loadu_si256(&rlo57.ymm[2]), xlo);
    gtlo3 = _mm256_cmpgt_epi64(_mm256_loadu_si256(&rlo57.ymm[3]), xlo);
    gtlo4 = _mm256_cmpgt_epi64(_mm256_loadu_si256(&rlo57.ymm[4]), xlo);
    gtlo0 = _mm256_and_si256(
        gtlo0, _mm256_cvtepi16_epi64(_mm256_castsi256_si128(eqhi)));
    gtlo1 = _mm256_and_si256(
        gtlo1, _mm256_cvtepi16_epi64(
                   _mm256_castsi256_si128(_mm256_bsrli_epi128(eqhi, 8))));
    eqm = _mm256_permute4x64_epi64(eqhi, 0xFF);
    gtlo2 = _mm256_and_si256(gtlo2, eqm); gtlo3 = _mm256_and_si256(gtlo3, eqm);
    gtlo4 = _mm256_and_si256(gtlo4, eqm); gtlo0 = _mm256_or_si256(gtlo0, gtlo1);
    gtlo0 = _mm256_add_epi64(_mm256_add_epi64(gtlo0, gtlo2),
                             _mm256_add_epi64(gtlo3, gtlo4));
    t = _mm_add_epi64(_mm256_castsi256_si128(gtlo0),
                      _mm256_extracti128_si256(gtlo0, 1));
    t = _mm_add_epi64(t, _mm_srli_si128(t, 8));
    r -= _mm_cvtsi128_si32(t);
    int32_t b = prng_next_u8(&ss->pc) & 1;
    *z_bimodal = b + ((b << 1) - 1) * r; *z_square = r * r;
}
```

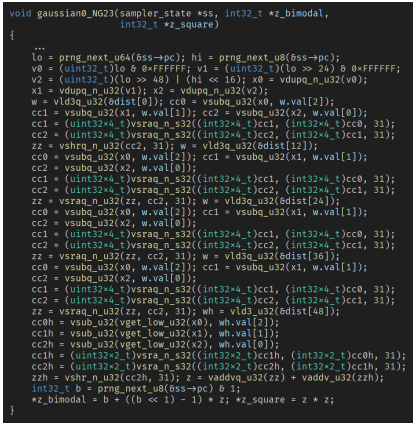
AVX2 impl. in NIST submission (72 bits = 15 + 57 bits; 44 cycles without PRNG overhead)

Vectorized BaseSampler: Previous Methods on ARMv8-A NEON



```c
void gaussian0_ref(sampler_state *ss, int32_t *z_bimodal,
                   int32_t *z_square)
{
    /* Get a random 72-bit value, into three 24-bit limbs (v0..v2). */
    uint64_t lo = prng_next_u64(&ss->pc);
    uint32_t hi = prng_next_u8(&ss->pc);
    uint32_t v0 = (uint32_t)lo & 0xFFFFFF;
    uint32_t v1 = (uint32_t)(lo >> 24) & 0xFFFFFF;
    uint32_t v2 = (uint32_t)(lo >> 48) | (hi << 16);
    /* Sampled value is z such that v0..v2 is lower than the first
    z elements of the table. */
    int32_t z = 0;
    for (size_t i = 0; i < (sizeof GAUSS0) / sizeof(GAUSS0[0]); i++) {
        uint32_t cc;
        cc = (v0 - GAUSS0[i][2]) >> 31;
        cc = (v1 - GAUSS0[i][1] - cc) >> 31;
        cc = (v2 - GAUSS0[i][0] - cc) >> 31;
        z += (int32_t)cc;
    }
    // Get a random bit b to turn the sampling into a bimodal distribution.
    int32_t b = prng_next_u8(&ss->pc) & 1;
    *z_bimodal = b + ((b << 1) - 1) * z;
    *z_square = z * z;
}
```

C impl. in `C-FN-DSA` (72 bits = 3×24 bits; 54 cycles without PRNG overhead)



```c
void gaussian0_NG23(sampler_state *ss, int32_t *z_bimodal,
                    int32_t *z_square)
{
    ...
    lo = prng_next_u64(&ss->pc); hi = prng_next_u8(&ss->pc);
    v0 = (uint32_t)lo & 0xFFFFFF; v1 = (uint32_t)(lo >> 24) & 0xFFFFFF;
    v2 = (uint32_t)(lo >> 48) | (hi << 16); x0 = vdupq_n_u32(v0);
    x1 = vdupq_n_u32(v1); x2 = vdupq_n_u32(v2);
    w = vld3q_u32(&dist[0]); cc0 = vsubq_u32(x0, w.val[2]);
    cc1 = vsubq_u32(x1, w.val[1]); cc2 = vsubq_u32(x2, w.val[0]);
    cc1 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc1, (int32x4_t)cc0, 31);
    cc2 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc2, (int32x4_t)cc1, 31);
    zz = vshrq_n_u32(cc2, 31); w = vld3q_u32(&dist[12]);
    cc0 = vsubq_u32(x0, w.val[2]); cc1 = vsubq_u32(x1, w.val[1]);
    cc2 = vsubq_u32(x2, w.val[0]);
    cc1 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc1, (int32x4_t)cc0, 31);
    cc2 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc2, (int32x4_t)cc1, 31);
    zz = vsraq_n_u32(zz, cc2, 31); w = vld3q_u32(&dist[24]);
    cc0 = vsubq_u32(x0, w.val[2]); cc1 = vsubq_u32(x1, w.val[1]);
    cc2 = vsubq_u32(x2, w.val[0]);
    cc1 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc1, (int32x4_t)cc0, 31);
    cc2 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc2, (int32x4_t)cc1, 31);
    zz = vsraq_n_u32(zz, cc2, 31); w = vld3q_u32(&dist[36]);
    cc0 = vsubq_u32(x0, w.val[2]); cc1 = vsubq_u32(x1, w.val[1]);
    cc2 = vsubq_u32(x2, w.val[0]);
    cc1 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc1, (int32x4_t)cc0, 31);
    cc2 = (uint32x4_t)vsraq_n_s32((int32x4_t)cc2, (int32x4_t)cc1, 31);
    zz = vsraq_n_u32(zz, cc2, 31); wh = vld3_u32(&dist[48]);
    cc0h = vsub_u32(vget_low_u32(x0), wh.val[2]);
    cc1h = vsub_u32(vget_low_u32(x1), wh.val[1]);
    cc2h = vsub_u32(vget_low_u32(x2), wh.val[0]);
    cc1h = (uint32x2_t)vsra_n_s32((int32x2_t)cc1h, (int32x2_t)cc0h, 31);
    cc2h = (uint32x2_t)vsra_n_s32((int32x2_t)cc2h, (int32x2_t)cc1h, 31);
    zzh = vshr_n_u32(cc2h, 31); z = vaddvq_u32(zz) + vaddv_u32(zzh);
    int32_t b = prng_next_u8(&ss->pc) & 1;
    *z_bimodal = b + ((b << 1) - 1) * z; *z_square = z * z;
}
```

NEON impl. in [NG23] (72 bits = 3×24 bits; 59 cycles without PRNG overhead)

Vectorized BaseSampler: The Strategy

Challenge: Strict KAT compatibility enforces sequential PRNG usage.

Our Solution:

1. **Relax KAT Compatibility:** Does *not* affect interoperability with verification.
2. **Batch Processing:** Generate N samples at once (e.g., $N = 16$ for AVX2).
3. **Modular Design:**
    - Main computation loop vectorized.
    - Bimodal transformation and squaring integrated.

Interoperability with Verification

The Trade-off

- Vectorization requires **parallel** PRNG usage.
- This breaks strict KAT compatibility (which mandates a sequential PRNG order).

Why Interoperability is Preserved?

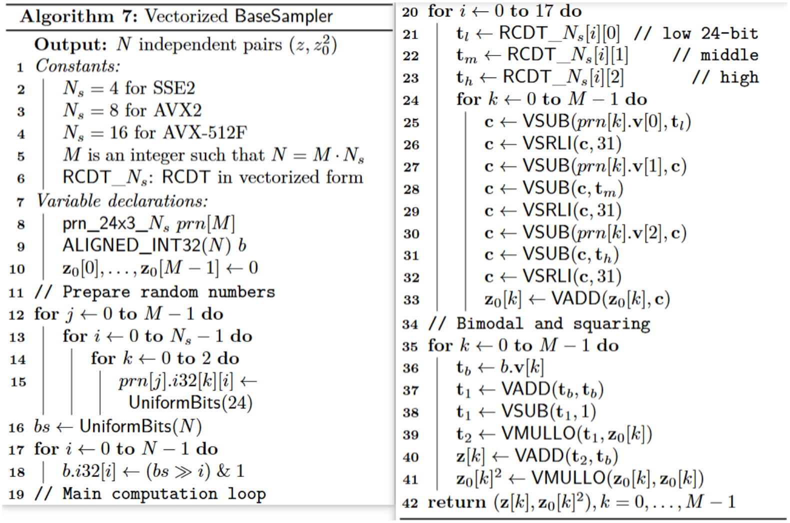
- Our modification does *not* alter the specific Gaussian distribution of the output vector s (line 7 of Alg. 1).
- It preserves the rejection sampling condition (line 8 of Alg. 1). The condition $\|s\|^2 \leq \lfloor \beta^2 \rfloor$ remains strictly satisfied.

** Note: The `C-FN-DSA` project employs a similar trade-off in its SHAKE256X4 variant.*

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

Figure: Alg. 1

Implementation on x86: Algorithm Overview



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
        $\mathbf{c} \leftarrow$ VSUB($prn[k].\mathbf{v}[0], \mathbf{t}_l$)
        $\mathbf{c} \leftarrow$ VSRLI($\mathbf{c}, 31$)
        $\mathbf{c} \leftarrow$ VSUB($prn[k].\mathbf{v}[1], \mathbf{c}$)
        $\mathbf{c} \leftarrow$ VSUB($\mathbf{c}, \mathbf{t}_m$)
        $\mathbf{c} \leftarrow$ VSRLI($\mathbf{c}, 31$)
        $\mathbf{c} \leftarrow$ VSUB($prn[k].\mathbf{v}[2], \mathbf{c}$)
        $\mathbf{c} \leftarrow$ VSUB($\mathbf{c}, \mathbf{t}_h$)
        $\mathbf{c} \leftarrow$ VSRLI($\mathbf{c}, 31$)
        $\mathbf{z}_0[k] \leftarrow$ VADD($\mathbf{z}_0[k], \mathbf{c}$)
// Bimodal and squaring
for $k \leftarrow 0$ to $M-1$ do
    $\mathbf{t}_b \leftarrow b.\mathbf{v}[k]$
    $\mathbf{t}_1 \leftarrow$ VADD($\mathbf{t}_b, \mathbf{t}_b$)
    $\mathbf{t}_1 \leftarrow$ VSUB($\mathbf{t}_1, 1$)
    $\mathbf{t}_2 \leftarrow$ VMULLO($\mathbf{t}_1, \mathbf{z}_0[k]$)
    $\mathbf{z}[k] \leftarrow$ VADD($\mathbf{t}_2, \mathbf{t}_b$)
    $\mathbf{z}_0[k]^2 \leftarrow$ VMULLO($\mathbf{z}_0[k], \mathbf{z}_0[k]$)
return $(\mathbf{z}[k], \mathbf{z}_0[k]^2), k = 0, \ldots, M-1$
```

Implementation: x86 (SSE2, AVX2, AVX-512F)

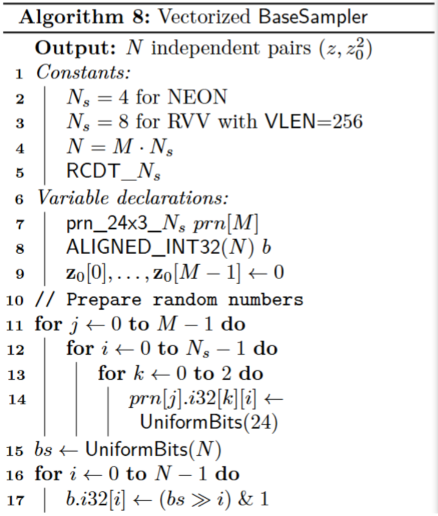
Comparison Optimization

- Reference: Requires subtraction with borrow on 3×24-bit integers.
- SIMD: Use `vpcmpgtd` (Compare Packed Signed Integers Greater Than).

Instruction Scheduling

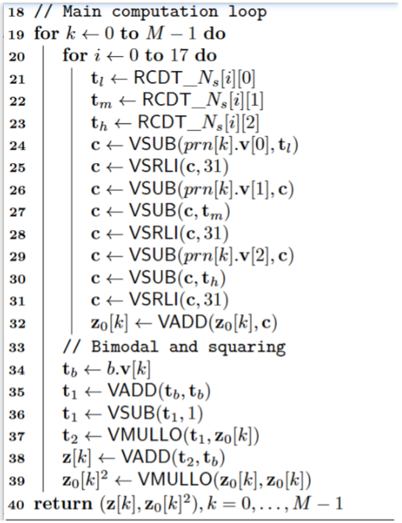
- **SSE2/AVX2:** Optimized using `vpcmpgtd` (Latency 1, CPI 0.5 on Rocket Lake).
- **AVX-512F:** Avoided `vpcmpgtd` due to higher latency (3 cycles).
- **Unrolling:** Fully unrolled inner loops (**for** $k \leftarrow 0$ **to** $M - 1$) to enable instruction interleaving and reduce pipeline stalls.

Implementation on RISC-V Vector (RVV): Algorithm Overview



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_$N_s[i][2]$
        $\mathbf{c} \leftarrow$ VSUB$(prn[k].\mathbf{v}[0], \mathbf{t}_l)$
        $\mathbf{c} \leftarrow$ VSRLI$(\mathbf{c}, 31)$
        $\mathbf{c} \leftarrow$ VSUB$(prn[k].\mathbf{v}[1], \mathbf{c})$
        $\mathbf{c} \leftarrow$ VSUB$(\mathbf{c}, \mathbf{t}_m)$
        $\mathbf{c} \leftarrow$ VSRLI$(\mathbf{c}, 31)$
        $\mathbf{c} \leftarrow$ VSUB$(prn[k].\mathbf{v}[2], \mathbf{c})$
        $\mathbf{c} \leftarrow$ VSUB$(\mathbf{c}, \mathbf{t}_h)$
        $\mathbf{c} \leftarrow$ VSRLI$(\mathbf{c}, 31)$
        $\mathbf{z}_0[k] \leftarrow$ VADD$(\mathbf{z}_0[k], \mathbf{c})$
    // Bimodal and squaring
    $\mathbf{t}_b \leftarrow b.\mathbf{v}[k]$
    $\mathbf{t}_1 \leftarrow$ VADD$(\mathbf{t}_b, \mathbf{t}_b)$
    $\mathbf{t}_1 \leftarrow$ VSUB$(\mathbf{t}_1, 1)$
    $\mathbf{t}_2 \leftarrow$ VMULLO$(\mathbf{t}_1, \mathbf{z}_0[k])$
    $\mathbf{z}[k] \leftarrow$ VADD$(\mathbf{t}_2, \mathbf{t}_b)$
    $\mathbf{z}_0[k]^2 \leftarrow$ VMULLO$(\mathbf{z}_0[k], \mathbf{z}_0[k])$
return $(\mathbf{z}[k], \mathbf{z}_0[k]^2), k = 0, \ldots, M-1$
```

Implementation: RISC-V Vector (RVV)

Constraints

- RVV comparison instructions output to mask registers, not usable directly in arithmetic.
- `vmsbc` (subtract with borrow) has dependency chains through mask register `v0`.

Optimization Strategy

- **Hybrid approach:** Pack 3 iterations into one macro.
- Use direct subtraction (`vsub`, `vsrl`) for 2 iterations + borrow method for 1 iteration.
- **Load-once-use-many:** Keep RCDT table in vector registers ($N = 64$).
  - $18 \times 3 = 54$ 24-bit integer segments, where 12 segments are zero-valued.
  - 24 scalar + 18 vector registers can hold 42 non-zero segments.
  - vx-type instructions: vsub.vx v2, v1, t0.

Implementation: ARMv8 NEON

Overview

- The implementation logic is **nearly identical to the RVV version**.
- Batch size: $N = 64$ (similar to RVV).

Register Allocation: Load-once-use-many

- *Challenge:* NEON lacks RVV's `.vx` feature (cannot use scalar registers as operands).
- *Strategy:*
    - Persistently store **20 RCDT table entries** in **20 vector registers**.
    - Load the remaining entries from memory on demand during execution.

Comparison Optimization

- Implemented using `cmgt` (Compare Greater Than).

Implementation: RV64IM

Strategy

- Designed for RISC-V processors **without RVV support**.
- Adopts the **Batch Processing** strategy ($N = 64$).
- Partition 72-bit integer into **High 8-bits** + **Low 64-bits**.

Load-once-use-many for RCDT table:

- **High 8-bits:** All 18 entries fit into 12-bit signed immediates (e.g., encoded directly in `addi`).
- **Low 64-bits:**
  - **2 entries** are $< 2^{11}$, encoded via immediates.
  - **remaining 16 entries** are stored in 16 registers.

```
for $j \leftarrow 0$ to $N$ do
    for $i \leftarrow 0$ to $17$ do
        // Set to 1 if less than
        $c \leftarrow$ SLTU$(prn[j].[0], \mathrm{RCDT}[i][0])$
        $c \leftarrow prn[j].[1] - c$
        $c \leftarrow c - \mathrm{RCDT}[i][1]$
        $c \leftarrow c \gg 63$
        $z_0[j] \leftarrow z_0[j] + c$
```

Figure: Part of Alg. 9

Vectorized FFT on RISC-V

Motivation: FFT is a major bottleneck on RISC-V (≈38%).

Key Innovation: Strided Load/Store

- RVV supports `vlse64.v` (Vector Load Strided Element).
- Minimal overhead compared to contiguous load (CPI 4 vs 3).
- Allows flexible coefficient loading for deep layer merging.

Layer Merging Strategies

- FALCON-512: **4+4** merging (vs NEON's 2+2+4 in [NG23]).
- FALCON-1024: **4+5** merging (vs NEON's 1+2+2+4 in [NG23]).
- Directly construct coefficient arrangements in registers, minimizing memory access.

Results: BaseSampler Performance

Speedup compared to Reference Implementation (`C-FN-DSA` project)

Instruction Set	Ref Cycles	Our Cycles	Speedup
SSE2	59	14	**4.2**×
AVX2	59	7	**8.4**×
AVX-512F	59	6	**9.8**×
NEON	54	30	**1.8**×
RVV	192	25	**7.7**×
RV64IM	192	51	**3.8**×

Note: The comparison excludes PRNG overhead to highlight sampler efficiency.

Results: FFT/iFFT on SpacemiT X60

Speedup compared to Reference Implementation (`C-FN-DSA` project)

Algorithm	**Impl.**	**Strategy**	**Cycles**	**Speedup**
FFT-1024	Ref	-	80,524	1.0×
	Our RV64D	**3+3+3**	**27,115**	**3.0**×
	Our RVV	**4+5**	**17,181**	**4.7**×
iFFT-1024	Ref	-	76,652	1.0×
	Our RV64D	**3+3+3**	**27,074**	**2.8**×
	Our RVV	**5+4**	**17,974**	**4.3**×

Results: FALCON-512 Signature Generation

Comparison with Reference Implementation (`C-FN-DSA` project)

Platform	Instruction Set	Ref (k)	Our (k)	Speedup
x86-64	SSE2	631	556	1.13×
	AVX2	543	441	1.23×
	AVX-512F	536	**393**	**1.36**×
ARMv8	NEON	1,230	1,053	1.17×
RISC-V	RV64GCB	2,535	1,867	1.36×
	RV64GCVB	2,530	**1,590**	**1.59**×

- **x86-64:** Significant gains from vectorized BaseSampler & 8-way Keccak (AVX-512).
- **RISC-V:** Massive speedup (1.59×) driven by RVV BaseSampler + FFT.
- *Note: Cycle counts in thousands (k).*

Conclusion

- We addressed the main bottlenecks in FALCON signature generation: `BaseSampler` and FFT.
- **Vectorized BaseSampler:** Implemented across 6 ISAs, yielding massive speedups (up to $9.8\times$).
- **RVV FFT:** Leveraged strided loads for 4+5 layer merging, achieving $> 4\times$ speedup.
- **Final Result:** Significant performance gains (up to $1.59\times$) for FALCON-$\{512,1024\}$ signature generation across x86, ARM, and RISC-V.

Paper: `https://eprint.iacr.org/2025/1867`
Artifact: `https://github.com/Ji-Peng/VecFalcon`
Slides: `https://ji-peng.github.io/uploads/tches2026/VecFalcon_slides.pdf`

Thank You!