ZEN

Zero Divisor **E**ncoding on **N**TRU Encryption

Algorithm Specification

(version 1.0.1)

October 28, 2026

Principal Submitter: Xianhui Lu[1,2], +86-15010131069, luxianhui@iie.ac.cn

Contributors:

- **Design:** Yijian Liu[1,2], liuyijian@iie.ac.cn
- **Analysis:** Dongshu Cai[1,2], caidongshu@iie.ac.cn

Dingding Jia[1,2], jiadingding@iie.ac.cn

Lei Bi[1,2], bilei@iie.ac.cn

Jingnan He[1,2], hejingnan@iie.ac.cn

- **Principal Implementation:** Yu Zhang[1,2], zhangyu1999@iie.ac.cn
- **ARM Implementation:** Zhe Liu[3], zhe.liu@zju.edu.cn

Junhao Huang[4], junhaohuang@smu.edu.sg

Lu Zhou[5], lu.zhou@nuaa.edu.cn

Liming Fang[6], flm@pqctech.cn

Tianhao Liu[3], lizyui@zju.edu.cn

Sihan He[5], h_sihan@nuaa.edu.cn

Meng Xu[7], 15210561626@163.com

- **FPGA Implementation:** Lutan Zhao[1,2], zhaolutan@iie.ac.cn

Xueqian Bao[1,2], baoxueqian@iie.ac.cn

Organizations: 1. State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS

2. University of Chinese Academy of Sciences

3. Zhe Jiang University

4. Singapore Management University

5. Nanjing University of Aeronautics and Astronautics

6. Hangzhou Post-quantum Cryptography Technology Co., Ltd

7. China Electric Power Research Institute Co., Ltd

Owner: Xianhui Lu

Postal Address: No. 19, Shu Cun Road, Beijing 100093, China

Contact: jiadingding@iie.ac.cn, +86-17788217158

Contents

1 Introduction **3**

1.1 NTRU Encryption Route 3

1.2 Positioning Relative to DAWN 4

2 Preliminary **5**

2.1 Ring, Polynomial, and Distribution 5

2.2 NTT, Multiplication, and Inversion 6

2.3 Storage, Compression, and Package 7

3 Algorithm **8**

3.1 Public Parameter 8

3.2 Algorithm Description 9

3.3 Correctness 11

4 Rationale **13**

5 Security **14**

5.1 Assumption 14

5.2 Security Reduction 15

5.3 DFR Analysis 17

5.4 Multi-target Security 20

5.5 Practical Security 21

6 Performance **22**

6.1 Parameter Selection 22

6.2 Protocol-level Usage 23

6.3 Implementation 23

6.3.1 Reference Implementation 23

6.3.2 Optimized Implementation on x86 with AVX2 24

6.3.3 Optimized Implementation on ARM Cortex-M3 25

6.3.4 Optimized Implementation on ARM Cortex-M4 26

6.3.5 Optimized Implementation on ARMv7 27

6.3.6 Optimized Implementation on ARMv8 28

6.3.7 Optimized Implementation on FPGA 30

7 Feature **31**

Changelog

Scope of this implementation update. This release updates only the x86 reference (plain C) and x86 AVX2 implementations of ZEN-128, ZEN-256, and ZEN-512. The corresponding changes to the ARM Cortex-M3, ARM Cortex-M4, ARMv7, ARMv8, and FPGA implementations are deferred to the next update. Descriptions and benchmark results for those platforms in the specification do not imply that they incorporate the fixes in this release.

DFR analysis. The revised analysis separates three kinds of DFRs: average-case on message δ_{ac}, worst-case on message δ_{ik}, and worst-case on key and message δ_{ZEN}. For consistency, both the security reduction and concrete security of DFR-based CCA attacks now use δ_{ZEN}.

Arithmetic and decoding implementations. The updated x86 implementations optimize polynomial inversion, NTT-domain arithmetic, and binary-polynomial operations, including batch-inversion, Karatsuba, and bit-packing techniques where applicable. Secret-indexed inversion-table accesses are replaced with fixed arithmetic sequences. SimpleDecoding writes both candidate correction positions at fixed addresses and selects their values with masks, while preserving the implementation's existing tie-breaking behavior.

Decapsulation hardening. Ciphertext verification and shared-secret selection use explicit unsigned masks. A GCC/Clang value barrier is added to prevent the observed compiler transformation of masked selection into a validation-dependent source-address selection. Verification of this change is specific to the tested compiler and target configurations; it is not a claim of a platform-independent constant-time proof.

Error propagation and failure handling. Fallible auxiliary-function calls are checked before their outputs are used, and failures propagate through the sampling, PKE, and KEM layers. Internal KEM failures return `ZEN_ERR_INTERNAL`, clear the relevant output buffers, and set their output lengths to zero. Key-generation retries are bounded before the sampling nonce can wrap around. Invalid ciphertexts continue to use implicit rejection; internal execution failures are reported separately.

Interfaces, builds, and symmetric backends. The AVX2 byte interfaces accept caller buffers of the documented lengths without extra alignment or SIMD tail padding. Separate `ngcc` and `performance` build profiles are provided. Separate symmetric-backend implementations and test vectors are also supplied; the alternative backend provides standard SHA-3/SHAKE instances for its 128- and 256-bit variants and a Keccak-extension variant for ZEN-512, with outputs evaluated separately from the SM3 backend.

Compatibility and validation. The x86 updates retain the existing SM3 parameter sets, sampling rules, public KEM function signatures, and serialized formats. Successful-path outputs are checked against the existing known-answer tests. Regression checks also cover valid and modified ciphertexts, buffer alignment, and injected auxiliary-function failures. Internal PKE helper interfaces now return status values where needed, and callers must check them.

1 Introduction

ZEN is submitted to the Next Generation Commercial Cryptography (NGCC) selection process as a DAWN-derived [28], NTRU-based key-encapsulation mechanism. The submission targets KEM-level IND-CCA2 security at the classical/quantum security-level pairs $(128, 80)$, $(256, 128)$, and $(512, 256)$. The parameter rows retained for these levels are recorded in Section 6. ZEN preserves the architectural backbone of DAWN, namely zero-divisor encoding and the double-encoding framework, while introducing changes at three levels. First, the algorithm updates the decryption; the group-based decryption failure rate (DFR) analysis in Section 5.3 makes the DFR estimate rigorous rather than heuristic. Second, the selected parameter rows use distributions that support direct sampling of coefficients from random bits. Third, the implementation is constant-time, includes an optimized polynomial inversion procedure, and covers multiple target platforms.

From a theoretical perspective, ZEN explores the boundary of compactness and efficiency under the prescribed security targets. The scheme uses well-studied algebraic structures to obtain high performance while remaining within the intended security border. Its design is centered on polynomial-ring arithmetic, leading to noise reduction and the correction of a small number of wrap errors, all of which contribute to compact ciphertexts and efficient computation.

From a practical perspective, ZEN is designed for deployment scenarios where bandwidth, storage, and execution time are the primary constraints. Its flexibility is limited by the use of a power-of-two cyclotomic ring, which makes it difficult to select natural parameter sets for 192-bit and 384-bit security levels. Nevertheless, at the supported security levels, ZEN offers considerable advantages in size and efficiency over other lattice-based KEMs.

1.1 NTRU Encryption Route

NTRU, proposed by Hoffstein, Pipher, and Silverman [21], is one of the earliest lattice-based encryption schemes. It remains attractive because encryption can be represented by a single ring element, giving NTRU a natural advantage in compactness over two-component Ring-LWE and Module-LWE ciphertexts. During the NIST post-quantum cryptography standardization process, two NTRU variants showed significant promise: NTRU Prime [6] advanced to the third round of candidates, while NTRU [11], originating from [21] and merging NTRU-HPS with NTRU-HRSS, advanced to the third round as a finalist. Although neither scheme was ultimately selected, Kyber [10], later standardized as ML-KEM [32], became the standardized lattice-based KEM. Nevertheless, the cryptographic community continues to explore and refine NTRU structures, with many recent improvements and variants [30, 16, 15, 41, 28] further improving the competitiveness of NTRU-based schemes.

Modern NTRU decryption methods can be broadly divided into two approaches to message extraction: algebraic encoding and geometric encoding. Most NTRU encryption schemes follow the original NTRU design [21], in which the message is encoded into a structured subspace, using integer encodings in the original construction and polynomial encodings in NEV [41] and DAWN [28]. The second approach extracts the message through the NTRU trapdoor without relying on an explicit encoding structure. BAT [16] is the main representative of this route;

however, generating the NTRU trapdoor is expensive, making key generation significantly slower than in many other schemes.

In the algebraic encoding approach, a central question is how to encode the message so that it can be recovered reliably in the presence of noise. The original NTRU encryption scheme uses a small positive integer, coprime to the modulus, to encode the message into the low bits, and many subsequent NTRU encryption schemes follow this method. However, this encoding amplifies the noise by the same integer factor. Higher noise leads to a larger modulus and larger sizes. Since KEM applications typically do not require the full available message space, lattice-based encryption schemes often provide redundancy beyond the security requirement; for example, a message space of size $2^{n/4}$ can already be sufficient. Hoffstein and Silverman therefore proposed a polynomial encoding method [22] that exploits the redundancy of message space to reduce noise amplification: by replacing the integer 3 with the polynomial $x + 2$, the amplification decreases from 3 to $\sqrt{5}$, at the cost of reducing the message space from 3^n to $2^n - 1$. Later, NEV [41] introduced vector encoding, replacing the integer 2 with a vector structure akin to D2/D4 codes; this reduces the amplification from 2 to $\sqrt{2}$, while reducing the message space from 2^n to $2^{n/4}$. More recently, DAWN [28] observed that replacing the polynomial encoding element with a zero-divisor polynomial exploits the message space more efficiently: the amplification decreases from 2 to $\sqrt{2}$, while the message space decreases from 2^n only to $2^{n/2}$. DAWN further exploits the algebraic structure by using the remaining $2^{n/4}$ space to construct a second encoding layer that corrects one wrap error under an independence assumption. ZEN builds on and extends the DAWN design.

1.2 Positioning Relative to DAWN

ZEN inherits the DAWN-style zero divisors $x^{n/2}+1$ and $x^{n/4}+1$, together with the same double encoding backbone. Under the submission rule excluding already standardized algorithms and variants without core modifications, the relevant distinction is that ZEN changes the underlying mechanisms rather than merely renaming DAWN components. DAWN provides the architectural baseline, while ZEN aims to make the construction more rigorous, simpler, and better suited for standardization.

At the algorithmic level, ZEN modifies two components of the DAWN-derived path. First, it modifies **SimpleDecoding** to enable analysis by quarter-dimensional groups. The DFR estimate in Section 5.3 is therefore formulated as a group-union upper bound over the induced four-dimensional group events. Second, the KEM layer follows an ML-KEM-style implicit rejection Fujisaki–Okamoto syntax: encapsulation derives both the accepted key and the encryption coins from the sampled message and the public-key hash, while decapsulation re-encrypts the recovered message and, on mismatch, falls back to a hash of the stored fallback secret and the ciphertext. This replaces the earlier Kyber-style presentation used around the DAWN baseline and improves the encapsulation/decapsulation path without changing the underlying PKE assumption basis.

At the parameter level, ZEN changes the distribution choices. The distributions used in the registered parameter rows are chosen so that sampling can be implemented using only additions, subtractions, and multiplications of bits, as described in Section 2.

At the implementation level, ZEN also changes the engineering target relative to the DAWN baseline. The implementation is designed to be constant-time by removing secret-dependent branches. The implementation of **FastInversion** is optimized to reduce inversion cost. The implementation effort also covers multiple platforms, including reference C, AVX2, ARM Cortex-M3, ARM Cortex-M4, ARMv7, ARMv8, and FPGA.

2 Preliminary

2.1 Ring, Polynomial, and Distribution

Throughout the document, n is a power of two and

$$\mathcal{R}_n = \mathbb{Z}[x]/(x^n+1), \qquad \mathcal{R}_{n,q} = \mathbb{Z}_q[x]/(x^n+1).$$

When it is helpful to make the quotient ideal explicit, reduction in $\mathcal{R}_{n,q}$ is written as

$$(\bmod\ \langle q, x^n+1\rangle).$$

For $a \in \mathbb{Z}$, write $a \pmod q \in (-\frac{q}{2}, \frac{q}{2}] \cap \mathbb{Z}$ for the centered representative. Unless stated otherwise, coefficients in $\mathbb{Z}_q$ are written with centered representatives. For a polynomial

$$\boldsymbol{a} = \sum_{i=0}^{n-1} \boldsymbol{a}_{[i]} x^i \in \mathcal{R}_n,$$

the coefficient of x^i is written $\boldsymbol{a}_{[i]}$, and

$$\|\boldsymbol{a}\|_\infty = \max_{0 \le i < n} |\boldsymbol{a}_{[i]}|.$$

The message space is

$$\mathcal{M} = \left\{ \boldsymbol{m}(x) = \sum_{i=0}^{n/4-1} \boldsymbol{m}_{[i]} x^i : \boldsymbol{m}_{[i]} \in \{0,1\} \right\}.$$

For $k \in \mathbb{N}$, let B_k denote the centered binomial distribution

$$B_k = \sum_{i=1}^{k} U_i - \sum_{i=1}^{k} V_i,$$

where the U_i and V_i are independent Bernoulli random variables with parameter $1/2$.

For $a \in (0, \frac{1}{2})$, let S_a denote the sparse ternary coefficient distribution

$$\Pr[S_a = 1] = \Pr[S_a = -1] = a, \qquad \Pr[S_a = 0] = 1 - 2a.$$

When two coefficient distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are written as $\mathcal{D}_1 + \mathcal{D}_2$, the sum is coefficient-wise, and the two draws are independent. In the sparse notation used by the approved rows, $2S_a$ denotes the law of $2X$ for $X \leftarrow S_a$. For any coefficient distribution $\mathcal{D}$, the notation $\mathcal{D}^n$ denotes

the polynomial distribution obtained by drawing n independent coefficients from $\mathcal{D}$.

The ZEN parameter rows in Section 6 use the family of distributions obtained from

$$B_1,\ B_2,\ B_3,\ S_{1/8},\ S_{1/16},\ S_{3/16},\ S_{3/32}$$

and their independent sums are listed in the parameter tables. We note that sampling from these sparse ternary distributions can be implemented using the same basic bit operations as for centered binomial distributions. Concretely,

$$S_{1/8} = (U_0 - U_1)U_2, \quad S_{1/16} = (U_0 - U_1)U_2U_3, \quad S_{3/16} = U_0U_1 - U_2U_3, \quad S_{3/32} = (U_0U_1 - U_2U_3)U_4$$

where the U_i are independent Bernoulli random variables with parameter $1/2$.

We use the function $\mathrm{Sample}(\mathcal{D})$ to denote the sampling process of an element drawn from the distribution $\mathcal{D}$, and we abuse the notion $\mathrm{Sample}(\mathcal{D}, \rho)$ to denote the sampling process with the given random seed ρ. We use $\mathcal{U}(S)$ to denote the uniform distribution over the finite set S.

2.2 NTT, Multiplication, and Inversion

ZEN uses the negacyclic Number Theoretic Transform (NTT) to accelerate multiplication in $\mathcal{R}_{n,q}$. Let d_{res} denote the residual degree after incomplete decomposition, where d_{res} is a power of two dividing n. Complete decomposition corresponds to $d_{\mathrm{res}} = 1$; incomplete NTT means $d_{\mathrm{res}} > 1$. Assume that q is prime and admits a primitive $(2n/d_{\mathrm{res}})$-th root of unity ζ, equivalently $q \equiv 1 \pmod{2n/d_{\mathrm{res}}}$. Then

$$\mathcal{R}_{n,q} \cong \prod_{0 \le i < n/d_{\mathrm{res}}} \mathbb{Z}_q[x]/(x^{d_{\mathrm{res}}} - \zeta^{2i+1}).$$

Concretely, the transform records the residues

$$\mathrm{NTT} : \mathcal{R}_{n,q} \to \prod_{0 \le i < n/d_{\mathrm{res}}} \mathbb{Z}_q[x]/(x^{d_{\mathrm{res}}} - \zeta^{2i+1}), \qquad \mathrm{NTT}(\boldsymbol{a}) = (\boldsymbol{a} \bmod (x^{d_{\mathrm{res}}} - \zeta^{2i+1}))_{0 \le i < n/d_{\mathrm{res}}}.$$

Multiplication is then performed component-wise in the residual rings, and the inverse transform is the Chinese-remainder recombination map satisfying

$$\mathrm{InverseNTT}(\mathrm{NTT}(\boldsymbol{a})) = \boldsymbol{a}.$$

For inversion, ZEN uses even/odd splitting before the residual degree becomes too large. Write

$$\boldsymbol{f}(x) = \boldsymbol{f}_0(x^2) + x\boldsymbol{f}_1(x^2),$$

where

$$\boldsymbol{f}_0(y) = \sum_{i=0}^{n/2-1} \boldsymbol{f}_{[2i]} y^i, \qquad \boldsymbol{f}_1(y) = \sum_{i=0}^{n/2-1} \boldsymbol{f}_{[2i+1]} y^i.$$

After the substitution $y = x^2$, the denominator $\boldsymbol{f}_0^2(y) - y\boldsymbol{f}_1^2(y)$ lives in the smaller ring $\mathbb{Z}_q[y]/(y^{n/2}+$

1). Hence

$$\boldsymbol{f}(x)^{-1} = \frac{\boldsymbol{f}_0(y) - x\boldsymbol{f}_1(y)}{\boldsymbol{f}_0^2(y) - y\boldsymbol{f}_1^2(y)}, \qquad y = x^2,$$

so the recursive inversion step is carried out in $\mathbb{Z}_q[y]/(y^{n/2}+1)$ rather than in the original degree-n ring. Repeating this reduction eventually reaches the complete-NTT case $d_{\mathrm{res}} = 1$.

2.3 Storage, Compression, and Package

To compactly store each element, we propose lossy polynomial compression and a package that preserves the compressed polynomials.

For a ciphertext alphabet of size M_c, define the deterministic tie-breaking rule

$$\mathrm{Round}_{\downarrow}(t) = \lfloor t + 1/2 \rfloor \qquad (t \in \mathbb{R}),$$

so half-integers are rounded upward. For $a \in \mathbb{Z}_q$ and $u \in \mathbb{Z}_{M_c}$, define

$$\begin{aligned} \mathrm{Compress}_{q,M_c}(a) &= \mathrm{Round}_{\downarrow}\left(\frac{M_c}{q}a\right) \pmod{M_c}, \\ \mathrm{Decompress}_{q,M_c}(u) &= \mathrm{Round}_{\downarrow}\left(\frac{q}{M_c}u\right) \pmod{q}. \end{aligned}$$

Both maps extend coefficient-wise to $\mathcal{R}_{n,q}$. When compression is applied to a polynomial $\boldsymbol{a}$, the induced compression error is written

$$\boldsymbol{e}_{q,M_c}(\boldsymbol{a}) = \mathrm{Decompress}_{q,M_c}(\mathrm{Compress}_{q,M_c}(\boldsymbol{a})) - \boldsymbol{a},$$

where subtraction uses centered representatives in $\mathbb{Z}_q$.

When $M_c = q$, the row is uncompressed, and the notation degenerates to

$$\mathrm{Compress}_{q,q} = \mathrm{Decompress}_{q,q} = \mathrm{id}, \qquad \boldsymbol{e}_{q,q}(\boldsymbol{a}) = \boldsymbol{0}.$$

This convention is used by the uncompressed ZEN rows.

For $N \in \{q, M_c\}$, let Pack_N and Unpack_N denote the fixed byte pack and unpack of coefficient vectors over $\mathbb{Z}_N$ without loss, such that

$$\mathrm{Unpack}_N(\mathrm{Pack}_N(\boldsymbol{a})) = \boldsymbol{a}.$$

Public keys and ciphertexts consist of polynomials whose coefficients are constrained to specified ranges. We view these polynomials as vectors and pack their coefficients strategically to optimize byte usage, thereby reducing storage requirements across the system.

For coefficient ranges that are powers of two, such as compressed ciphertext polynomials, we use direct bit concatenation. For the remaining coefficients, which are taken modulo 769, we follow the packing strategy of NEV [41] and DAWN [28]: each block of five coefficients in $\mathbb{Z}_{769}$ is encoded into 48 bits and then split into 6 bytes.

3 Algorithm

ZEN works in the power-of-two cyclotomic ring $\mathcal{R}_{n,q}$ and uses the NTRU public-key relation

$$\boldsymbol{h} \equiv \boldsymbol{g}\boldsymbol{f}^{-1} \pmod{\langle q, x^n + 1\rangle},$$

and derives a KEM from a PKE core through FO-style layering with implicit rejection [28, 17, 23, 26]. ZEN uses the DAWN-style double-encoding path. It uses the canonical lift $\frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m}$ from Section 2, retains the decoding structure based on $x^{n/2} + 1$ and $x^{n/4} + 1$, and serializes ciphertexts through the explicit coefficient-compression map. Figure 1 summarizes this structure.

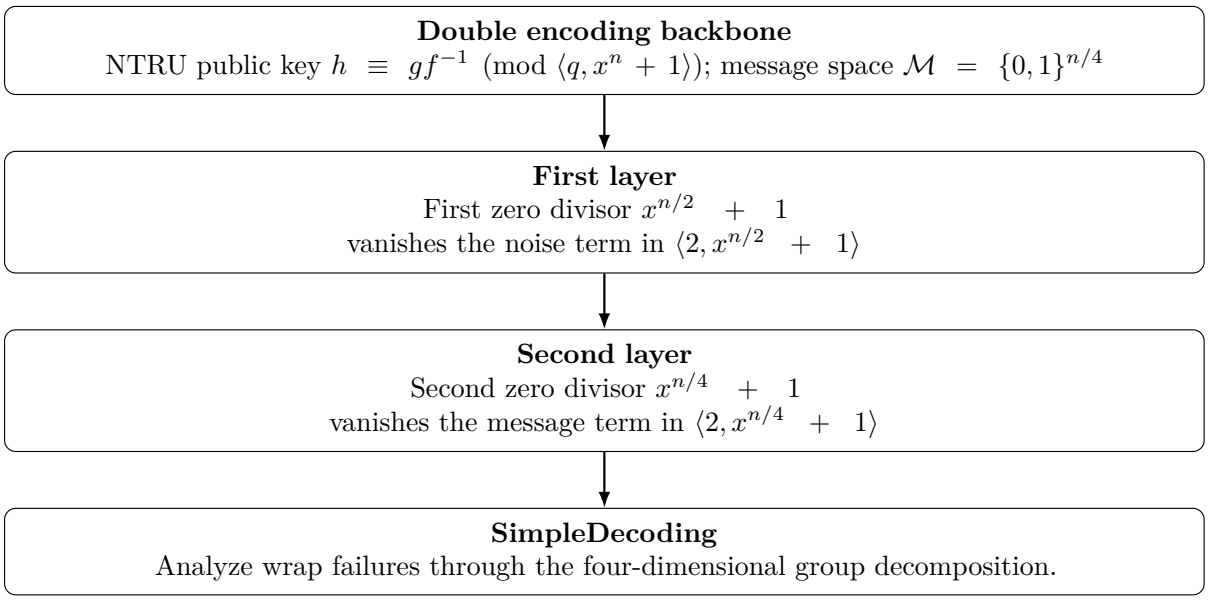


Figure 1: Two-layer design structure of ZEN.

3.1 Public Parameter

Every ZEN parameter row fixes the ring degree n, the modulus q, the sampling distributions attached to $\boldsymbol{g}$, $\boldsymbol{f}$, $\boldsymbol{s}$, and $\boldsymbol{e}$, and the parameters corresponding to compression. In summary, we have the following public parameters:

- $n \in \mathbb{N}$, the degree, a power-of-2
- $q \in \mathbb{N}$, the modulus, a prime integer
- $\mathcal{D}_g, \mathcal{D}_f, \mathcal{D}_s, \mathcal{D}_e$, the distributions of the coefficients of $\boldsymbol{g}$, $\boldsymbol{f}$, $\boldsymbol{s}$, and $\boldsymbol{e}$
- $M_c \in \{128, 256, 512, q\}$, the compressed ciphertext size
- $\mathcal{M}$, the message space $\{0,1\}^{n/4}$

3.2 Algorithm Description

Following the DAWN presentation, ZEN is stated in two layers. The underlying public-key encryption layer is given first; the KEM layer then wraps that PKE through the standard FO transform with implicit rejection. ZEN instantiates the FO-layer functions H_m, H_{pk}, and H_K with domain-separated SM3-based hash functions. The canonical encoders and decoders are the coefficient packers used for the public-key and ciphertext payload sizes reported in Section 6.

Auxiliary Algorithms We present the auxiliary algorithms used in ZEN. **FastInversion** is used during key generation to accelerate polynomial inversion in $\mathbb{Z}_2[x]/(x^n+1)$, leveraging the algebraic relation $x^n + 1 \equiv (x+1)^n \pmod 2$. **SimpleDecoding** is used during decryption to correct errors at the group level using simple operations.

Algorithm 1 FastInversion$(\boldsymbol{f}, d)$

Require: polynomial $\boldsymbol{f}$, degree d
Ensure: inverse $\boldsymbol{f}_{\text{inv}}$ or $\perp$
if $\boldsymbol{f} \equiv 0 \pmod{\langle x+1, 2\rangle}$ **then**
  return $\perp$
end if
$l \leftarrow \log_2(d)$
$\boldsymbol{k} \leftarrow (\boldsymbol{f}+1)/(x+1)$
$\boldsymbol{f}_{\text{inv}} \leftarrow 1$
for $i = 0$ to $l-1$ **do**
  $\boldsymbol{b} \leftarrow \boldsymbol{k}\boldsymbol{f}_{\text{inv}} \pmod{\langle x^{2^i}+1, 2\rangle}$
  $\boldsymbol{k} \leftarrow \boldsymbol{k} + \boldsymbol{f}\boldsymbol{b} \pmod{\langle x^n+1, 2\rangle}$
  $\boldsymbol{k} \leftarrow \boldsymbol{k}/(x^{2^i}+1)$
  $\boldsymbol{f}_{\text{inv}} \leftarrow \boldsymbol{f}_{\text{inv}} + (x^{2^i}+1)\boldsymbol{b}$
end for
return $\boldsymbol{f}_{\text{inv}}$

Algorithm 2 SimpleDecoding$(\boldsymbol{a}, \boldsymbol{m}', \boldsymbol{f}_{\text{inv}})$

Require: polynomials $\boldsymbol{a}$, $\boldsymbol{m}'$, $\boldsymbol{f}_{\text{inv}}$
Ensure: polynomial $\boldsymbol{m}$
$\boldsymbol{\delta} \leftarrow \boldsymbol{a} \pmod{\langle 2, x^{n/4}+1\rangle}$
$\boldsymbol{\eta} \leftarrow 0$
for $i = 0$ to $n/4-1$ **do**
  $v_0 \leftarrow \min\left(\left||\boldsymbol{a}_{[i]}| - \frac{q}{2}\right|, \left||\boldsymbol{a}_{[i+n/2]}| - \frac{q}{2}\right|\right)$
  $v_1 \leftarrow \min\left(\left||\boldsymbol{a}_{[i+n/4]}| - \frac{q}{2}\right|, \left||\boldsymbol{a}_{[i+3n/4]}| - \frac{q}{2}\right|\right)$
  if $v_0 \le v_1$ **then**
    $\boldsymbol{\eta} \leftarrow \boldsymbol{\eta} + x^i\boldsymbol{\delta}_{[i]}$
  else
    $\boldsymbol{\eta} \leftarrow \boldsymbol{\eta} + x^{i+n/4}\boldsymbol{\delta}_{[i]}$
  end if
end for
$\boldsymbol{m} \leftarrow \left(\boldsymbol{m}' + \boldsymbol{\eta}\boldsymbol{f}_{\text{inv}} \pmod{\langle 2, x^{n/2}+1\rangle}\right) \pmod{\langle 2, x^{n/4}\rangle}$
return $\boldsymbol{m}$

Algorithms of ZEN.PKE ZEN.PKE is built from the double-encoding skeleton. Its algorithms are specified in Algorithms 3, 4, and 5.

Algorithm 3 ZEN.PKE.KeyGen(ρ)

Require: seeds ρ
Ensure: public key pk, secret key sk_{PKE}
$N \leftarrow 0$
repeat
  $\boldsymbol{f} \leftarrow \mathrm{Sample}(\mathcal{D}_f^n, \mathrm{PRF}(\rho, N))$
  $\boldsymbol{f}_{\mathrm{inv}} \leftarrow \textbf{FastInversion}(\boldsymbol{f}, n/2)$
  $\widehat{\boldsymbol{f}} \leftarrow \mathrm{NTT}(\boldsymbol{f})$
  $N \leftarrow N + 1$
until $\prod_{0 \le i < 256} \widehat{\boldsymbol{f}}_{[i]} \neq 0$ and $\boldsymbol{f}_{\mathrm{inv}} \neq \perp$
repeat
  $\boldsymbol{g} \leftarrow \mathrm{Sample}(\mathcal{D}_g^n, \mathrm{PRF}(\rho, N))$
  $\widehat{\boldsymbol{g}} \leftarrow \mathrm{NTT}(\boldsymbol{g})$
  $N \leftarrow N + 1$
until $\prod_{0 \le i < 256} \widehat{\boldsymbol{g}}_{[i]} \neq 0$
$\widehat{\boldsymbol{h}} \leftarrow \widehat{\boldsymbol{g}} \odot \widehat{\boldsymbol{f}}^{-1} \pmod{\langle q, x^n + 1 \rangle}$
$pk \leftarrow \mathrm{Pack}_q(\widehat{\boldsymbol{h}})$
$sk_{\mathrm{PKE}} \leftarrow (\widehat{\boldsymbol{f}}, \boldsymbol{f}_{\mathrm{inv}})$
return (pk, sk_{PKE})

Algorithm 4 ZEN.PKE.Encrypt$(pk, \boldsymbol{m}, \rho)$

Require: public key pk, message $\boldsymbol{m}$, seed ρ
Ensure: ciphertext ct
$\widehat{\boldsymbol{h}} \leftarrow \mathrm{Unpack}_q(pk)$
$\boldsymbol{s} \leftarrow \mathrm{Sample}(\mathcal{D}_s^n, \mathrm{PRF}(\rho, 0))$
$\boldsymbol{e} \leftarrow \mathrm{Sample}(\mathcal{D}_e^n, \mathrm{PRF}(\rho, 1))$
$\widehat{\boldsymbol{s}} \leftarrow \mathrm{NTT}(\boldsymbol{s})$
$\boldsymbol{u} \leftarrow \mathrm{InverseNTT}(\widehat{\boldsymbol{h}} \odot \widehat{\boldsymbol{s}}) + \boldsymbol{e} \pmod{\langle q, x^n + 1 \rangle}$
$\boldsymbol{c} \leftarrow \boldsymbol{u} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m} \pmod{\langle q, x^n + 1 \rangle}$
$ct \leftarrow \mathrm{Pack}_{M_c}\Big(\mathrm{Compress}_{q,M_c}(\boldsymbol{c})\Big)$
return ct

Algorithm 5 ZEN.PKE.Decrypt(sk_{PKE}, ct)

Require: secret key sk_{PKE}, ciphertext ct
Ensure: message $\boldsymbol{m}$
Parse sk_{PKE} as $(\widehat{\boldsymbol{f}}, \boldsymbol{f}_{\mathrm{inv}})$
$\widetilde{\boldsymbol{c}} \leftarrow \mathrm{Unpack}_{M_c}(ct)$
$\boldsymbol{c}^{\dagger} \leftarrow \mathrm{Decompress}_{q,M_c}(\widetilde{\boldsymbol{c}})$
$\widehat{\boldsymbol{c}} \leftarrow \mathrm{NTT}(\boldsymbol{c}^{\dagger})$
$\boldsymbol{a} \leftarrow (x^{n/2} + 1)\,\mathrm{InverseNTT}(\widehat{\boldsymbol{c}} \odot \widehat{\boldsymbol{f}}) \pmod{\langle q, x^n + 1 \rangle}$
$\boldsymbol{m}' \leftarrow \boldsymbol{a}\boldsymbol{f}_{\mathrm{inv}} \pmod{\langle 2, x^{n/2} + 1 \rangle}$
$\boldsymbol{m} \leftarrow \textbf{SimpleDecoding}(\boldsymbol{a}, \boldsymbol{m}', \boldsymbol{f}_{\mathrm{inv}})$
return $\boldsymbol{m}$

Algorithms of ZEN.KEM ZEN.KEM is obtained from ZEN.PKE using an ML-KEM-style implicit-rejection Fujisaki–Okamoto transform. During encapsulation, the message and the public-key hash are hashed together to derive the encryption coins and the shared key. During decapsulation, the recovered message is re-encrypted; if the resulting ciphertext does not match the input ciphertext, decapsulation instead outputs a hash of the stored fallback secret and the ciphertext. The algorithms are specified in Algorithms 6, 7, and 8.

Algorithm 6 ZEN.KEM.KeyGen(ρ_g, ρ_f)

Require: seeds ρ_g, ρ_f
Ensure: public key pk, secret key sk_{KEM}
1: $(pk, sk_{\mathrm{PKE}}) \leftarrow$ **ZEN.PKE.KeyGen**(ρ_g, ρ_f)
2: $\kappa' \leftarrow \mathrm{Sample}(\mathcal{U}(\{0,1\}^{n/4}))$
3: $h_{\mathrm{pk}} \leftarrow \mathsf{H}_{pk}(pk)$
4: $sk_{\mathrm{KEM}} \leftarrow (sk_{\mathrm{PKE}}, pk, h_{\mathrm{pk}}, \kappa')$
5: **return** (pk, sk_{KEM})

Algorithm 7 ZEN.KEM.Encaps(pk)

Require: public key pk
Ensure: shared key K, ciphertext ct
1: $\boldsymbol{m} \leftarrow \mathrm{Sample}(\mathcal{U}(\{0,1\}^{n/4}))$
2: $(K, \rho) \leftarrow \mathsf{H}_m(\boldsymbol{m}, \mathsf{H}_{pk}(pk))$
3: $ct \leftarrow$ **ZEN.PKE.Encrypt**$(pk, \boldsymbol{m}, \rho)$
4: **return** (K, ct)

Algorithm 8 ZEN.KEM.Decaps(sk_{KEM}, ct)

Require: secret key sk_{KEM}, ciphertext ct
Ensure: shared key K or K'
1: $(sk_{\mathrm{PKE}}, pk, h_{\mathrm{pk}}, \kappa') \leftarrow sk_{\mathrm{KEM}}$
2: $\boldsymbol{m} \leftarrow$ **ZEN.PKE.Decrypt**(sk_{PKE}, ct)
3: $(K, \rho) \leftarrow \mathsf{H}_m(\boldsymbol{m}, h_{\mathrm{pk}})$
4: $K' \leftarrow \mathsf{H}_K(\kappa', ct)$
5: $ct' \leftarrow$ **ZEN.PKE.Encrypt**$(pk, \boldsymbol{m}, \rho)$
6: **if** $ct' = ct$ **then**
7: **return** K
8: **else**
9: **return** K'
10: **end if**

3.3 Correctness

This subsection expands the decryption path in Algorithm 5. The purpose is to make explicit the two binary decoding layers behind **SimpleDecoding**, following the same DAWN-style double-encoding viewpoint [28].

Let the decompressed ciphertext polynomial be

$$\boldsymbol{c}^{\dagger} = \boldsymbol{h}\boldsymbol{s} + \boldsymbol{e} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m} + \boldsymbol{e}_{q,M_c}(\boldsymbol{c}),$$

where $\boldsymbol{e}_{q,M_c}(\boldsymbol{c})$ is the deterministic compression error. Decryption first computes

$$\boldsymbol{a} \equiv (x^{n/2}+1)\boldsymbol{f}\boldsymbol{c}^{\dagger} \pmod{\langle q, x^n+1\rangle}.$$

Using $\boldsymbol{h}\boldsymbol{f} \equiv \boldsymbol{g} \pmod{\langle q, x^n+1\rangle}$, this value satisfies

$$\boldsymbol{a} \equiv x^{n/2}(x^{n/4}+1)\boldsymbol{f}\boldsymbol{m} + \boldsymbol{\varepsilon}_{\text{wrap}} \pmod{\langle q, x^n+1\rangle},$$

where

$$\boldsymbol{\varepsilon}_{\text{wrap}} = (x^{n/2}+1)\left(\boldsymbol{g}\boldsymbol{s} + \boldsymbol{f}\boldsymbol{e} + \boldsymbol{f}\boldsymbol{e}_{q,M_c}(\boldsymbol{c})\right).$$

Set the wrap argument

$$\boldsymbol{z}_{\text{wrap}} = x^{n/2}(x^{n/4}+1)\boldsymbol{f}\boldsymbol{m} + \boldsymbol{\varepsilon}_{\text{wrap}} \in \mathbb{Z}[x]/(x^n+1)$$

and define the coefficientwise q-wrap error

$$\boldsymbol{e}_{\text{wrap}} = \left\lfloor \frac{1}{q} \cdot \boldsymbol{z}_{\text{wrap}} \right\rceil.$$

The centered lift of $\boldsymbol{a}$ is then

$$\boldsymbol{z}_{\text{wrap}} - q \cdot \boldsymbol{e}_{\text{wrap}}.$$

Since q is odd, $\boldsymbol{f}\boldsymbol{f}_{\text{inv}} \equiv 1$ modulo $\langle 2, x^{n/2}+1\rangle$, and $\boldsymbol{\varepsilon}_{\text{wrap}}$ has the factor $x^{n/2}+1$, the first binary decoding layer gives

$$\begin{aligned}\boldsymbol{m}' &\equiv \boldsymbol{a}\boldsymbol{f}_{\text{inv}} \pmod{\langle 2, x^{n/2}+1\rangle} \\ &\equiv (x^{n/4}+1)\boldsymbol{m} + \boldsymbol{e}_{\text{wrap}}\boldsymbol{f}_{\text{inv}} \pmod{\langle 2, x^{n/2}+1\rangle}.\end{aligned}$$

Thus the remaining task is to recover the binary representative of the wrap vector $\boldsymbol{e}_{\text{wrap}}$ in the quotient $\mathbb{F}_2[x]/(x^{n/2}+1)$.

The second binary layer is the **SimpleDecoding** step. Let $\nu = n/4$, and group the four coefficients

$$G_r = \{r,\ r+\nu,\ r+2\nu,\ r+3\nu\}, \qquad 0 \le r < \nu.$$

Write the four wrap bits in group G_r as

$$B_{r,j} = (\boldsymbol{e}_{\text{wrap}})_{[r+j\nu]} \bmod 2, \qquad j \in \{0,1,2,3\}.$$

Reducing modulo $\langle 2, x^{n/4}+1\rangle$ identifies all four positions in G_r. The syndrome coefficient used by **SimpleDecoding** is therefore

$$W_r = B_{r,0} \oplus B_{r,1} \oplus B_{r,2} \oplus B_{r,3}.$$

Message recovery, however, is performed modulo $\langle 2, x^{n/2}+1\rangle$, where the group splits into two half-cycles. The two target bits are

$$T_{r,0} = B_{r,0} \oplus B_{r,2}, \qquad T_{r,1} = B_{r,1} \oplus B_{r,3}.$$

They satisfy $W_r = T_{r,0} \oplus T_{r,1}$. If $W_r = 0$, the decoder sets both target bits to zero. If $W_r = 1$, it uses the centered q-level residues of $\boldsymbol{a}$ to decide which half-cycle contains the odd wrap:

$$S_{r,0} = \min\left(\left||\boldsymbol{a}_{[r]}| - \frac{q}{2}\right|, \left||\boldsymbol{a}_{[r+2\nu]}| - \frac{q}{2}\right|\right),$$

$$S_{r,1} = \min\left(\left||\boldsymbol{a}_{[r+\nu]}| - \frac{q}{2}\right|, \left||\boldsymbol{a}_{[r+3\nu]}| - \frac{q}{2}\right|\right).$$

If $S_{r,0} \leq S_{r,1}$, the recovered representative has $(\widehat{T}_{r,0}, \widehat{T}_{r,1}) = (1, 0)$; if $S_{r,1} < S_{r,0}$, it has $(\widehat{T}_{r,0}, \widehat{T}_{r,1}) = (0, 1)$. Ties are resolved in favor of the first half-cycle, consistently with Algorithm 2. The DFR analysis in Section 5.3 distinguishes the exact decoding failure event from its conservative upper bound. Collecting these recovered half-cycle bits over all r gives the polynomial $\boldsymbol{\eta}$, which is the decoder's estimate of $\boldsymbol{e}_{\text{wrap}}$ modulo $\langle 2, x^{n/2} + 1\rangle$.

A sufficient condition for correct decoding in each group G_r is

$$|z_{\text{wrap}}[r + i\nu]| + |z_{\text{wrap}}[r + j\nu]| < q, \quad \forall\, i \in \{0, 2\},\ j \in \{1, 3\}.$$

Under this condition, nonzero wrap bits cannot occur in both half-cycles. If $W_r = 0$, then $(T_{r,0}, T_{r,1}) = (0, 0)$. If $W_r = 1$, exactly one wrap bit is nonzero, and its centered residue has strictly larger absolute value than every residue in the opposite half-cycle. Hence the comparison between $S_{r,0}$ and $S_{r,1}$ identifies the correct half-cycle. Consequently, SimpleDecoding recovers the correct wrap representative whenever the above condition holds for every group. This condition is sufficient but not necessary; Section 5.3 bounds the probability of its violation using the projected-tail analysis. After $\boldsymbol{e}_{\text{wrap}}$ is recovered, decryption cancels the wrap term:

$$\boldsymbol{m}' + \boldsymbol{e}_{\text{wrap}} \boldsymbol{f}_{\text{inv}} \equiv (x^{n/4} + 1)\boldsymbol{m} \pmod{\langle 2, x^{n/2} + 1\rangle}.$$

The final reduction modulo $\langle 2, x^{n/4}\rangle$ keeps the first copy of the double encoding and outputs $\boldsymbol{m}$.

4 Rationale

Our core philosophy follows a standard design principle: additional algebraic structure improves practicality, but may also increase cryptanalytic risk. Since NTRU has been studied for decades, we believe its risks are comparatively well understood and that its structure can be used effectively under careful control.

NTRU, Zero Divisors, and Double Encoding We use an NTRU-like structure as the skeleton for two main reasons: long-term public scrutiny and structural compactness. Since the original NTRU scheme [21] was proposed in 1996, NTRU-type constructions have been extensively analyzed. The hardness of the NTRU assumption has also been related, through reductions, to GapSVP over ideal lattices [39, 40]. From another perspective, NTRU can be viewed as a homogeneous form of RLWE. Regarding concrete security, the dense sublattice feature of NTRU has been well studied in lattice attacks [14]; moreover, NTRU-style hybrid attacks predate the recent rotation based enhanced hybrid attacks on MLWE [33, 24]. For

compactness, NTRU-style public key encryption represents a ciphertext as a single ring element, whereas RLWE-style public-key encryption typically requires two ring elements.

We use zero divisor encoding rather than classical integer encoding because the message space in practical KEM usage contains redundancy. The natural polynomial structure of zero divisors allows them to be embedded into encryption and decryption in a simple and algebraically compatible way, also with the reduction to the equal hardness.

Furthermore, in the common lattice-based PKE schemes, the message space occupies only a quarter-dimensional subspace. Using a zero divisor to mitigate noise amplification consumes only half of the dimension, leaving a quarter dimensional redundancy. We therefore use double encoding as the primary mechanism to exploit this remaining message space. The second layer uses another zero divisor, chosen to be a divisor of the zero divisor used in the first layer, thereby yielding a natural group-level error-correction mechanism.

By combining NTRU, zero divisor encoding, and double encoding, ZEN exploits algebraic structure at multiple levels while keeping the resulting security assumptions and attack surface under explicit control.

Distribution and Sampling For ease of implementation and fast sampling, ZEN uses only distributions that can be sampled using bit additions, bit subtractions, bit multiplication, or combinations of these operations. In particular, centered binomial distribution is one of these distributions. These distributions are flexible and allow parameters to be selected conveniently to balance size, correctness, and security.

Ring and Modulus We use a power-of-two cyclotomic ring as the base ring because it supports efficient NTT-based arithmetic and has been studied extensively in lattice-based cryptography. Recent analyses of module-BKZ also indicate that power-of-two cyclotomic rings provide comparatively strong resistance against module-BKZ. For the modulus, we balance size and efficiency. Since an NTRU public key cannot be compressed directly in the same way as some RLWE-style keys, a smaller modulus is important for reducing key size. We therefore choose the smallest modulus that supports incomplete NTT while preserving the targeted security level and decryption correctness.

5 Security

5.1 Assumption

This section works in the ring $\mathcal{R}_{n,q} = \mathbb{Z}_q[x]/(x^n + 1)$ and uses the same algorithmic notation as Section 3. The assumption basis follows the standard background on NTRU and Ring-LWE over power-of-two cyclotomic rings [28, 21, 39, 35, 29].

Definition 1 (Decisional NTRU)**.** *Let $\boldsymbol{f} \leftarrow \mathcal{D}_f^n$ and $\boldsymbol{g} \leftarrow \mathcal{D}_g^n$, with $\boldsymbol{f}$ invertible in $\mathcal{R}_{n,q}$. The decisional NTRU problem is to distinguish $\boldsymbol{h} = \boldsymbol{g}\boldsymbol{f}^{-1} \pmod{\langle q, x^n + 1\rangle}$ from a uniform element of $\mathcal{R}_{n,q}$. For an adversary $\mathcal{A}$, the corresponding advantage is written $\mathsf{Adv}^{\mathsf{dNTRU}}_{n,q,\mathcal{D}_f,\mathcal{D}_g}(\mathcal{A})$.*

Definition 2 (Decisional Ring-LWE)**.** *Let* $\boldsymbol{a} \leftarrow U(\mathcal{R}_{n,q})$*,* $\boldsymbol{s} \leftarrow \mathcal{D}_s^n$*,* $\boldsymbol{e} \leftarrow \mathcal{D}_e^n$*, and* $\boldsymbol{r} \leftarrow U(\mathcal{R}_{n,q})$*. The decisional Ring-LWE problem is to distinguish* $(\boldsymbol{a}, \boldsymbol{as}+\boldsymbol{e})$ *from* $(\boldsymbol{a}, \boldsymbol{r})$*. For an adversary* $\mathcal{A}$*, the corresponding advantage is written* $\mathsf{Adv}^{\mathsf{dRLWE}}_{n,q,\mathcal{D}_s,\mathcal{D}_e}(\mathcal{A})$*.*

Definition 3 (Security notions and oracle convention)**.** *For a public-key encryption scheme,* $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}$ *denotes the usual chosen-plaintext distinguishing advantage.*

For a KEM, $\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}$ *denotes the standard adaptive chosen-ciphertext advantage against the decapsulation oracle, excluding the challenge ciphertext. The FO-style KEM reductions below model* H_m*,* H_{pk}*, and* H_K *as random oracles;* q_H *denotes the total number of random-oracle queries and* q_D *the number of classical decapsulation queries [17, 23, 26].*

5.2 Security Reduction

Throughout this subsection, let the message space be $\mathcal{M} = \{0,1\}^{n/4}$.

IND-CPA security of the PKE layer. An encryption of $\boldsymbol{m}$ computes

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{hs} + \boldsymbol{e} \pmod{\langle q, x^n+1\rangle}, \\ ct &= \mathrm{Pack}_{M_c}\Big(\mathrm{Compress}_{q,M_c}\Big(\boldsymbol{u} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m}\Big)\Big), \end{aligned}$$

where $\boldsymbol{s} \leftarrow \mathcal{D}_s^n$ and $\boldsymbol{e} \leftarrow \mathcal{D}_e^n$.

Theorem 1 (IND-CPA security of ZEN.PKE)**.** *For every* $\mathsf{IND\text{-}CPA}$ *adversary* $\mathcal{A}$ *against* ZEN.PKE*, there exist adversaries* $\mathcal{B}_{\mathsf{N}}$ *and* $\mathcal{B}_{\mathsf{L}}$*, with running times comparable to that of* $\mathcal{A}$*, such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathrm{ZEN.PKE}}(\mathcal{A}) \le \mathsf{Adv}^{\mathsf{dNTRU}}_{n,q,\mathcal{D}_f,\mathcal{D}_g}(\mathcal{B}_{\mathsf{N}}) + \mathsf{Adv}^{\mathsf{dRLWE}}_{n,q,\mathcal{D}_s,\mathcal{D}_e}(\mathcal{B}_{\mathsf{L}}).$$

Proof. The proof follows the standard DAWN hybrid argument applied to the ZEN.PKE ciphertext syntax.

Game 0. This is the real $\mathsf{IND\text{-}CPA}$ game for ZEN.PKE. The public key is $\boldsymbol{h} = \boldsymbol{gf}^{-1}$, and the challenge ciphertext is

$$ct^* = \mathrm{Pack}_{M_c}\Big(\mathrm{Compress}_{q,M_c}\Big(\boldsymbol{u} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m}_b\Big)\Big)$$

for $\boldsymbol{u} = \boldsymbol{hs} + \boldsymbol{e}$ and the hidden challenge bit b.

Game 1. Replace the NTRU public key $\boldsymbol{h} = \boldsymbol{gf}^{-1}$ by a uniform element of $\mathcal{R}_{n,q}$. A distinguisher between Game 0 and Game 1 immediately gives a distinguisher for Definition 1, so

$$|\Pr[\mathsf{G}_0 \Rightarrow 1] - \Pr[\mathsf{G}_1 \Rightarrow 1]| \le \mathsf{Adv}^{\mathsf{dNTRU}}_{n,q,\mathcal{D}_f,\mathcal{D}_g}(\mathcal{B}_{\mathsf{N}}).$$

Game 2. Keep $\boldsymbol{h}$ uniform and replace the Ring-LWE term $\boldsymbol{hs}+\boldsymbol{e}$ by a uniform $\boldsymbol{r} \leftarrow U(\mathcal{R}_{n,q})$. The challenge ciphertext becomes

$$ct^* = \mathrm{Pack}_{M_c}\Big(\mathrm{Compress}_{q,M_c}\Big(\boldsymbol{r} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m}_b\Big)\Big).$$

A distinguisher between Game 1 and Game 2 gives a distinguisher for Definition 2, hence

$$|\Pr[\mathsf{G}_1 \Rightarrow 1] - \Pr[\mathsf{G}_2 \Rightarrow 1]| \leq \mathsf{Adv}^{\mathsf{dRLWE}}_{n,q,\mathcal{D}_s,\mathcal{D}_e}(\mathcal{B}_{\mathsf{L}}).$$

In Game 2, the polynomial

$$\boldsymbol{r} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m}_b$$

is uniform in $\mathcal{R}_{n,q}$ for either value of b, because addition by a fixed ring element is a permutation of $\mathcal{R}_{n,q}$. The packing map $\mathrm{Pack}_{M_c} \odot \mathrm{Compress}_{q,M_c}$ is deterministic, so applying it to a uniform ring element produces a ciphertext distribution independent of b. Therefore, the adversary has zero advantage in Game 2. Summing the two hybrid gaps proves the theorem. □

Decryption-failure probabilities. For (pk, sk_{PKE}) output by $\mathsf{ZEN.PKE.KeyGen}$, let

$$p_{pk,sk_{\mathrm{PKE}}}(\boldsymbol{m}) = \Pr_{\rho}[\mathsf{ZEN.PKE.Decrypt}(sk_{\mathrm{PKE}}, \mathsf{ZEN.PKE.Encrypt}(pk, \boldsymbol{m}, \rho)) \neq \boldsymbol{m}].$$

We distinguish the average-message DFR, the worst-case independent-key DFR, and the worst-case DFR:

$$\begin{aligned}
\delta_{\mathrm{ac}} &= \mathbb{E}_{(pk,sk_{\mathrm{PKE}})}\mathbb{E}_{\boldsymbol{m}\leftarrow U(\mathcal{M})}p_{pk,sk_{\mathrm{PKE}}}(\boldsymbol{m}),\\
\delta_{\mathrm{ik}} &= \max_{\boldsymbol{m}\in\mathcal{M}} \mathbb{E}_{(pk,sk_{\mathrm{PKE}})}p_{pk,sk_{\mathrm{PKE}}}(\boldsymbol{m}),\\
\delta_{\mathrm{ZEN}} &= \mathbb{E}_{(pk,sk_{\mathrm{PKE}})} \max_{\boldsymbol{m}\in\mathcal{M}} p_{pk,sk_{\mathrm{PKE}}}(\boldsymbol{m}).
\end{aligned}$$

Thus $\delta_{\mathrm{ac}} \leq \delta_{\mathrm{ik}} \leq \delta_{\mathrm{ZEN}}$. The maximum in δ_{ik} is outside key generation, whereas δ_{ZEN} allows the message to depend on the generated key. The FO reduction in Theorem 2 assumes worst-case correctness for the actual ZEN.PKE; the numerical results in Section 5.3 use a separate reference model and do not by themselves establish this assumption. A treatment based on δ_{ik} additionally requires analysis of failure finding dependent on the target public key.

The KEM layer is the implicit-rejection Fujisaki–Okamoto transform applied to ZEN.PKE. Let $\mathcal{K}_0$ denote the session key space used by the FO layer. In Algorithm 7, the accepted shared key is returned by H_m, while q_{H_K} counts fallback-oracle queries to H_K.

Theorem 2 (IND-CCA2 security of ZEN.KEM)**.** *Assume that ZEN.PKE has worst-case correctness error at most δ_{ZEN} and let $\mathcal{A}$ be an* IND-CCA2 *adversary against ZEN.KEM making $q_{\mathsf{H}_{pk}}, q_{\mathsf{H}_K}, q_{\mathsf{H}_m}$ quantum random-oracle queries as above. Under the remaining hypotheses of the cited implicit-rejection FO reductions in the QROM [23, 26], there exists an* IND-CPA *adversary $\mathcal{B}$ against ZEN.PKE such that*

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}_{\mathrm{ZEN.KEM}}(\mathcal{A}) \leq\ & 2(q_{\mathsf{H}_{pk}} + q_{\mathsf{H}_K})\sqrt{\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathrm{ZEN.PKE}}(\mathcal{B})}\\
&+ \frac{11q_{\mathsf{H}_K}}{\sqrt{|\mathcal{K}_0|}} + \frac{2q_{\mathsf{H}_{pk}} + 2q_{\mathsf{H}_m}}{\sqrt{|\mathcal{M}|}} + 4q_{\mathsf{H}_{pk}}\sqrt{\delta_{\mathrm{ZEN}}}.
\end{aligned}$$

Proof. Algorithms 6–8 instantiate the usual implicit-rejection FO syntax: encapsulation samples $\boldsymbol{m} \leftarrow \mathcal{M}$, derives deterministic encryption coins and an accepted key from $(\boldsymbol{m}, \mathsf{H}_{pk}(pk))$, and

decapsulation decrypts, re-encrypts, and falls back to $\mathsf{H}_K(\kappa', ct)$ on mismatch. This matches the intended implicit-rejection syntax; the bound above remains conditional on the precise correctness and random-oracle hypotheses of the cited reduction. The square-root term is the standard measure-and-reprogram loss for recovering the message query underlying the challenge encapsulation; the remaining two terms bound oracle guessing/collision events and correctness failures in the decapsulation simulation. Combining this theorem with Theorem 1 yields the full assumption-to-KEM reduction path for ZEN. □

5.3 DFR Analysis

Lemma 1 (Exact algebraic correctness implication for ZEN)**.** *Let* $\boldsymbol{u} = \boldsymbol{h}\boldsymbol{s} + \boldsymbol{e} \pmod{\langle q, x^n + 1\rangle}$ *be the pre-message ciphertext body, and let*

$$\boldsymbol{c}^\dagger = \boldsymbol{u} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m} + \boldsymbol{e}_{q,M_c}(\boldsymbol{c})$$

be the decompressed ciphertext polynomial used by Algorithm 5. During decryption, the algorithm computes $\boldsymbol{a} \equiv (x^{n/2}+1)\boldsymbol{f}\boldsymbol{c}^\dagger \pmod{\langle q, x^n+1\rangle}$*. Using* $\boldsymbol{h}\boldsymbol{f} \equiv \boldsymbol{g} \pmod{\langle q, x^n+1\rangle}$ *and the identity* $(1 + x^{n/2})(1 + x^{n/4} + x^{n/2} + x^{3n/4}) = 2x^{n/2}(1 + x^{n/4})$ *in* $\mathbb{Z}[x]/(x^n + 1)$*, the decoding relation is*

$$\boldsymbol{a} \equiv x^{n/2}(x^{n/4} + 1)\boldsymbol{f}\boldsymbol{m} + \boldsymbol{\varepsilon}_{\mathrm{wrap}} \pmod{\langle q, x^n + 1\rangle},$$

where

$$\boldsymbol{\varepsilon}_{\mathrm{wrap}} = (x^{n/2} + 1)\left(\boldsymbol{g}\boldsymbol{s} + \boldsymbol{f}\boldsymbol{e} + \boldsymbol{f}\boldsymbol{e}_{q,M_c}(\boldsymbol{c})\right).$$

Define the integer wrap argument

$$\boldsymbol{z}_{\mathrm{wrap}} = x^{n/2}(x^{n/4} + 1)\boldsymbol{f}\boldsymbol{m} + \boldsymbol{\varepsilon}_{\mathrm{wrap}} \in \mathbb{Z}[x]/(x^n + 1),$$

and its coefficientwise nearest-q quotient $\boldsymbol{e}_{\mathrm{wrap}} = \lfloor \boldsymbol{z}_{\mathrm{wrap}}/q \rceil$ *in* $\mathbb{Z}[x]/(x^n+1)$*. Then, after centered lifting, multiplying by* $\boldsymbol{f}_{\mathrm{inv}}$*, and reducing modulo* $\langle 2, x^{n/2}+1\rangle$*, the intermediate value* $\boldsymbol{m}'$ *satisfies*

$$\boldsymbol{m}' \equiv (x^{n/4} + 1)\boldsymbol{m} + \boldsymbol{e}_{\mathrm{wrap}}\boldsymbol{f}_{\mathrm{inv}} \pmod{\langle 2, x^{n/2} + 1\rangle}.$$

Consequently, if the induced binary wrap representative is decoded correctly by the DAWN-style wrap decoder associated with $x^{n/4} + 1$*, decryption outputs* $\boldsymbol{m}$*.*

Proof. Using the definition of $\boldsymbol{c}^\dagger$,

$$\begin{aligned}\boldsymbol{a} &\equiv (x^{n/2} + 1)\boldsymbol{f}\left(\boldsymbol{h}\boldsymbol{s} + \boldsymbol{e} + \frac{q+1}{2}(1 + x^{n/4} + x^{n/2} + x^{3n/4})\boldsymbol{m} + \boldsymbol{e}_{q,M_c}(\boldsymbol{c})\right)\\ &\equiv (x^{n/2} + 1)(\boldsymbol{g}\boldsymbol{s} + \boldsymbol{f}\boldsymbol{e} + \boldsymbol{f}\boldsymbol{e}_{q,M_c}(\boldsymbol{c})) + x^{n/2}(x^{n/4} + 1)\boldsymbol{f}\boldsymbol{m} \pmod{\langle q, x^n + 1\rangle}.\end{aligned}$$

Hence $\boldsymbol{a} \equiv \boldsymbol{z}_{\mathrm{wrap}} \pmod{\langle q, x^n + 1\rangle}$, and its centered lift is $\boldsymbol{z}_{\mathrm{wrap}} - q\boldsymbol{e}_{\mathrm{wrap}}$. Multiplying by $\boldsymbol{f}_{\mathrm{inv}}$ and reducing modulo $\langle 2, x^{n/2} + 1\rangle$ gives

$$\boldsymbol{m}' \equiv \left((x^{n/4} + 1)\boldsymbol{f}\boldsymbol{m} + \boldsymbol{\varepsilon}_{\mathrm{wrap}} - q\boldsymbol{e}_{\mathrm{wrap}}\right)\boldsymbol{f}_{\mathrm{inv}} \pmod{\langle 2, x^{n/2} + 1\rangle}.$$

Since

$$\boldsymbol{f}\boldsymbol{f}_{\text{inv}} \equiv 1 \pmod{\langle 2, x^{n/2}+1\rangle},$$

since q is odd, and since

$$\boldsymbol{\varepsilon}_{\text{wrap}} = (x^{n/2}+1)\,(\boldsymbol{g}\boldsymbol{s} + \boldsymbol{f}\boldsymbol{e} + \boldsymbol{f}\boldsymbol{e}_{q,M_c}(\boldsymbol{c})) \equiv 0 \pmod{\langle 2, x^{n/2}+1\rangle},$$

we obtain

$$\boldsymbol{m}' \equiv (x^{n/4}+1)\boldsymbol{m} + \boldsymbol{e}_{\text{wrap}}\boldsymbol{f}_{\text{inv}} \pmod{\langle 2, x^{n/2}+1\rangle}.$$

□

Numerical probability model. The compression error $\boldsymbol{e}_{q,M_c}(\boldsymbol{c})$ in ZEN.PKE.Encrypt is a deterministic function of the uncompressed ciphertext and is generally correlated with $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{s}, \boldsymbol{e}$, and $\boldsymbol{m}$. For the three compressed main parameter sets, the numerical calculations below use a reference model in which its coefficients are independent of these variables and of one another, with probabilities 256/769 for each of $-1, 0, 1$ and 1/769 for 2. The prescribed coefficient distributions and the invertibility checks of ZEN.PKE.KeyGen are retained. The joint four-dimensional blocks and the shared factor $\boldsymbol{f}$ in $\boldsymbol{f}\boldsymbol{U}$ and $\boldsymbol{f}\boldsymbol{m}$ are also retained. These reference-model estimates do not by themselves bound the DFR of the actual deterministic compressor.

Four-dimensional group decomposition. The group decomposition preserves the dependencies among the coefficients of $\boldsymbol{z}_{\text{wrap}}$. It follows the coefficient-grouping approach of the multidimensional DFR analysis of Paiva et al. [34], using $\mu = 4$ in the negacyclic ring below. Set

$$\nu = \frac{n}{4}, \qquad Y = x^{\nu} = x^{n/4}.$$

Since the ambient ring is $\mathbb{Z}[x]/(x^n+1)$, we have $Y^4 = x^n = -1$. Thus each quarter-cycle group is naturally represented in the small negacyclic ring $\mathbb{Z}[Y]/(Y^4+1)$. For a polynomial $\boldsymbol{a} \in \mathbb{Z}[x]/(x^n+1)$, define its r-th $\mu = 4$ block by

$$\text{block}_r(\boldsymbol{a}) = \boldsymbol{a}_{[r]} + \boldsymbol{a}_{[r+\nu]}Y + \boldsymbol{a}_{[r+2\nu]}Y^2 + \boldsymbol{a}_{[r+3\nu]}Y^3 \in \mathbb{Z}[Y]/(Y^4+1), \qquad 0 \le r < \nu.$$

The fixed polynomials of ZEN become $x^{n/2}+1 = 1+Y^2$ and $x^{n/4}+1 = 1+Y$. Write $\tau = 1+Y^2$ and $\omega = 1+Y$. Hence $x^{n/2}+1$ and $x^{n/4}+1$ are not treated as generic sparse multipliers; they are part of the four-dimensional algebraic block structure.

For two polynomials

$$\boldsymbol{a} = \sum_{u=0}^{\nu-1} x^u \,\text{block}_u(\boldsymbol{a}), \qquad \boldsymbol{b} = \sum_{u=0}^{\nu-1} x^u \,\text{block}_u(\boldsymbol{b}),$$

their product block satisfies

$$\text{block}_r(\boldsymbol{a}\boldsymbol{b}) = \sum_{u=0}^{\nu-1} Y^{\epsilon_r(u)} \,\text{block}_u(\boldsymbol{a})\,\text{block}_{(r-u) \bmod \nu}(\boldsymbol{b}),$$

where the carry exponent is

$$\epsilon_r(u) = \begin{cases} 0, & u \le r, \\ 1, & u > r. \end{cases}$$

This identity is the $\mu = 4$ negacyclic group splitting used in the DFR calculation. Let

$$\boldsymbol{U} = \boldsymbol{e} + \boldsymbol{e}_{q,M_c}(\boldsymbol{c}).$$

Then the r-th four-dimensional block of the wrap argument can be written as

$$\mathrm{block}_r(\boldsymbol{z}_{\mathrm{wrap}}) = \sum_{u=0}^{\nu-1} Y^{\epsilon_r(u)} \left[\tau G_u S_v + F_u(\tau U_v + Y^2 \omega M_v)\right], \qquad v = (r-u) \bmod \nu,$$

where

$$G_u = \mathrm{block}_u(\boldsymbol{g}), \quad S_v = \mathrm{block}_v(\boldsymbol{s}), \quad F_u = \mathrm{block}_u(\boldsymbol{f}), \quad U_v = \mathrm{block}_v(\boldsymbol{U}), \quad M_v = \mathrm{block}_v(\boldsymbol{m}).$$

This expression preserves the dependence between the four coefficients in the same quarter-cycle group and also preserves the correlation caused by the shared factor $\boldsymbol{f}$ in $\boldsymbol{fU}$ and $\boldsymbol{fm}$.

Exact group-level failure event. For each $0 \le r < \nu$, write $\mathrm{block}_r(\boldsymbol{z}_{\mathrm{wrap}}) = Z_{r,0} + Z_{r,1}Y + Z_{r,2}Y^2 + Z_{r,3}Y^3$. Let $a_{r,i} = \mathrm{Center}_q(Z_{r,i})$ and $B_{r,i} = (Z_{r,i} - a_{r,i})/q \bmod 2$ for $i \in \{0,1,2,3\}$. The true half-cycle wrap parities are $T_{r,0} = B_{r,0} \oplus B_{r,2}$ and $T_{r,1} = B_{r,1} \oplus B_{r,3}$, with syndrome $W_r = T_{r,0} \oplus T_{r,1}$.

The **SimpleDecoding** rule outputs $(\widehat{T}_{r,0}, \widehat{T}_{r,1}) = (0,0)$ when $W_r = 0$. When $W_r = 1$, it compares $\max(|a_{r,0}|, |a_{r,2}|)$ with $\max(|a_{r,1}|, |a_{r,3}|)$ and selects $(1,0)$ if the former is greater than or equal to the latter, and $(0,1)$ otherwise. Thus the exact group error event is $\mathsf{Bad}_r = \{(\widehat{T}_{r,0}, \widehat{T}_{r,1}) \ne (T_{r,0}, T_{r,1})\}$. In particular, a tie does not necessarily constitute a decoding failure under this deterministic rule. Nevertheless, ties occurring when $W_r = 1$ are covered by the conservative projected-tail upper bound below.

Let $P_r^{(4)}$ be the joint distribution of $(Z_{r,0}, Z_{r,1}, Z_{r,2}, Z_{r,3})$ in the specified key/message experiment. The exact group failure probability is $p_r^{(4)} = \Pr[\mathsf{Bad}_r]$, obtained by summing $P_r^{(4)}$ over this event. No independence between the four coordinates is assumed, and the union bound $\Pr[\bigcup_{r=0}^{\nu-1} \mathsf{Bad}_r] \le \sum_{r=0}^{\nu-1} p_r^{(4)}$ does not require independence between groups. When the group distribution is stationary, the latter bound is $\nu p_0^{(4)}$.

Projected-tail upper bound. Evaluating the full four-dimensional distribution can be expensive. A useful relaxation bounds Bad_r by the event that $|Z_{r,i}| + |Z_{r,j}| \ge q$ for some $i \in \{0,2\}$ and $j \in \{1,3\}$. For each such pair,

$$\Pr[|Z_{r,i}| + |Z_{r,j}| \ge q] \le \sum_{\sigma,\tau \in \{\pm 1\}} \Pr[\sigma Z_{r,i} + \tau Z_{r,j} \ge q].$$

Each term is a one-dimensional projected tail. In the pre-rejection independent-coefficient reference model, the four-dimensional block splitting yields a convolution of local projected laws.

For uniform independent message bits, or for a constant all-zero or all-one message, write

$$D_{\boldsymbol{\lambda},c} = \mathsf{Law}\Big(\Big\langle \boldsymbol{\lambda}, Y^c \left(\tau GS + F(\tau U + Y^2 \omega M)\right)\Big\rangle\Big), \qquad c \in \{0,1\},$$

where G, F, S, U, M denote independent input blocks except that the same F occurs in both summands involving F. For the projected group score $L_{r,\boldsymbol{\lambda}} = \langle \boldsymbol{\lambda}, \mathrm{block}_r(\boldsymbol{z}_{\mathrm{wrap}})\rangle$, this gives

$$\mathsf{Law}(L_{r,\boldsymbol{\lambda}}) = D_{\boldsymbol{\lambda},0}^{*(r+1)} * D_{\boldsymbol{\lambda},1}^{*(\nu-r-1)}.$$

For an arbitrary fixed message, use the corresponding bit-specific local laws instead. Conditioning on the invertibility tests also couples the secret blocks; the identities above are the pre-rejection convolution kernels, with key conditioning accounted for separately in the probability bounds. Writing $\boldsymbol{e}_i$ for the standard basis vector of $\mathbb{Z}^4$, the projected bound for a specified experiment is

$$\Pr\left[\bigcup_{r=0}^{\nu-1} \mathsf{Bad}_r\right] \le \sum_{r=0}^{\nu-1} p_r^{(4)} \le \sum_{r=0}^{\nu-1} \sum_{\substack{i\in\{0,2\}\\ j\in\{1,3\}}} \sum_{\sigma,\tau\in\{\pm 1\}} \Pr\Big[L_{r,\sigma\boldsymbol{e}_i+\tau\boldsymbol{e}_j} \ge q\Big].$$

The DFR bounds in Table 2 incorporate the exact wrap geometry, the prescribed key-invertibility conditioning, and the projected-tail calculation. They apply to the reference probability model specified above; the deterministic compressor requires separate analysis of its correlated quantization error.

5.4 Multi-target Security

Standard IND-CPA and IND-CCA notions are usually formulated for a single public key and a single challenge ciphertext. In practical deployments, however, practical security may be affected by multi-target settings. We distinguish two common cases: multi-public-key security and multi-ciphertext security. In the former, the adversary is given many public keys and succeeds if it recover any one of the private keys. In the latter, the adversary is given many ciphertexts, typically under a long-term public key, and succeeds if it learns the plaintext or shared key associated with any one of them.

Multi-public-key security is particularly relevant to decryption-failure attacks. We follow the FO transform used in ML-KEM [32] to mitigate multi-public-key attacks. Concretely, the public key hash is included when deriving both the shared key and the encryption randomness. This binds the derived randomness to the public key and prevents literal reuse of the same hash-derived candidate ciphertexts across different keys; it does not exclude general multi-target preprocessing or amortization.

Multi-ciphertext security can also affect practical security, especially when many ciphertexts are generated under the same long-term key and the message space is relatively small. Given many target ciphertexts, an adversary may try to find a collision with a precomputed table of encapsulations; the generic success probability scales roughly as the product of the number of precomputed trials and the number of target ciphertexts, divided by the message-space size. According to FrodoKEM [19], KEMs based on the FO transform can obtain stronger multi-

ciphertext security by using the salted FO transform [18], at the cost of appending a short salt to the ciphertext. ZEN uses the unsalted FO transform, without an additional salt field in the ciphertext.

5.5 Practical Security

A KEM requires security analysis of both the private key and the shared key. The private-key analysis is based on decisional NTRU, while the shared-key analysis is based on decisional Ring-LWE. We estimate practical security for both Ring-LWE and NTRU. The RLWE analysis includes lattice attacks, BKW, algebraic attacks, and hybrid attacks. The NTRU analysis has no dual attack [2], but it includes the dense-sublattice attack proposed in [14].

We employ the lattice estimator [1] of [3] to derive practical security parameters for ZEN. This tool provides strength estimates for both Ring-LWE and NTRU instantiations. For each attack strategy, we compute its computational cost and adopt the minimum value as the practical security level. Practical security is assessed along two axes. First, the estimator is run under both the `MATZOV` and `CoreSVP` (ADPS16) BKZ cost models. In this document, `CoreSVP` is treated as the more conservative model, since it suppresses polynomial factors when computing BKZ costs. Second, the security estimate for the shared key remains conservative by ignoring the additional noise introduced by compression. Although ZEN's security does not rely on the RLWR assumption, such compression would make attacks more difficult in practice. The parameter table also includes backup profiles with reduced or no compression. Table 1 records the estimates of the practical security, grouped by problem family and cost model as computed by the lattice estimator.

The lattice estimator identifies `dual_hybrid` as the most effective attack among those considered. Its complexity estimates follow the heuristic models of [20, 31], whose accuracy has been examined critically in [13]. Provable analyses of dual attacks [36, 37] give higher complexity estimates. We use the lattice estimator results when selecting the security parameters.

Table 1: Computed practical security for ZEN

(n, q)	compression	BKZ model	NTRU classical	NTRU quantum	RLWE classical	RLWE quantum
$(512, 769)$	$769 \to 128$	`MATZOV`	139.4	133.6	141.7	136.3
		`CoreSVP`	118.6	107.6	118.6	109.9
$(512, 769)$	$769 \to 256$	`MATZOV`	148.7	141.9	152.0	145.4
		`CoreSVP`	128.5	116.6	129.4	118.5
$(1024, 769)$	$769 \to 256$	`MATZOV`	272.0	252.9	280.9	263.8
		`CoreSVP`	257.5	233.7	261.9	240.4
$(1024, 769)$	$769 \to 769$	`MATZOV`	272.0	252.9	280.9	263.8
		`CoreSVP`	257.5	233.7	261.9	240.4
$(2048, 769)$	$769 \to 256$	`MATZOV`	530.5	485.4	537.5	496.7
		`CoreSVP`	526.2	477.5	519.9	477.9
$(2048, 769)$	$769 \to 769$	`MATZOV`	530.5	485.4	537.5	496.7
		`CoreSVP`	526.2	477.5	519.9	477.9

Failure-based CCA attacks. Following the failure-boosting framework of D'Anvers et al.[2], we account for generic offline filtering of candidate inputs. Motivated by directional failure

[1] `https://github.com/malb/lattice-estimator/`, with the latest commit `3e48ef4`

[2] `https://lirias.kuleuven.be/retrieve/548770`

boosting[3], we conservatively treat the first decryption failure as a successful attack, without charging for subsequent failures or key recovery. If an α fraction of candidates is selected with average conditional failure probability β, the modeled costs are $W_C = 1/(\alpha\beta)$ and $W_Q = 1/(\sqrt{\alpha}\,\beta)$, where the quantum speedup applies only to generic offline filtering and decapsulation queries remain classical.

For a fixed $\boldsymbol{m}$, let $p_{\boldsymbol{m}}(\boldsymbol{s}, \boldsymbol{U})$ be its conditional failure probability over accepted keys under the reference model in Section 5.3. We bound $M_r = \max_{\boldsymbol{m}\in\mathcal{M}} \mathbb{E}_{\boldsymbol{s},\boldsymbol{U}}[p_{\boldsymbol{m}}(\boldsymbol{s}, \boldsymbol{U})^r]$. Since $\alpha\beta^r \le M_r$, these moment bounds also cover mixtures of predeclared fixed messages and filtering thresholds. With $q_D \le 2^{80}$ and success probability at least $1/2$, we obtain $M_3 < 2^{-301}$, $M_4 < 2^{-532}$, and $M_{17} < 2^{-1821}$ for ZEN-128, ZEN-256, and ZEN-512, respectively. The resulting generic first-failure work lower bounds appear in Table 2.

Table 2: Reference-model DFR upper bounds and generic work lower bounds for the first failure ($q_D \le 2^{80}$, success probability at least $1/2$).

Scheme	δ_{ac}	δ_{ik}	δ_{ZEN}	Classical work	Quantum work
ZEN-128	$< 2^{-128.08}$	$< 2^{-120.34}$	$< 2^{-99.94}$	$> 2^{138}$	$> 2^{109}$
ZEN-256	$< 2^{-174.74}$	$< 2^{-160.39}$	$< 2^{-127.28}$	$> 2^{288}$	$> 2^{184}$
ZEN-512	$< 2^{-179.13}$	$< 2^{-157.96}$	$< 2^{-106.36}$	$> 2^{524}$	$> 2^{302}$

The first two DFR quantities are tightly bounded within the stated reference model, while the worst-case DFR bounds are less tight. The work columns are lower bounds in an independent-key, static generic first-failure model, not measured attack costs or full IND-CCA2 security levels. They do not cover arbitrary target-public-key-dependent filtering, adaptive use of decapsulation feedback, or multi-target amortization. In addition, the independent quantization model does not establish these bounds for actual deterministic compression. Consequently, the work bounds cannot replace the correctness hypothesis of Theorem 2.

Additionally, to mitigate timing attacks, we eliminate secret-dependent branching throughout our implementation. For conditional operations such as if/else statements, we employ masking techniques that execute all possible branches and use a mask variable to select the correct result. This approach ensures that execution time remains independent of secret data. To validate our constant-time implementation, we employ the dynamic analysis tool TIMECOP[4], which is included in the SUPERCOP benchmarking suite, to detect potential timing vulnerabilities. Our analysis confirms that the program contains no conditional jumps or move operations dependent on secret data, thereby ensuring protection against timing side-channel attacks.

6 Performance

6.1 Parameter Selection

ZEN targets 128/256/512 classical security bits and 80/128/256 quantum security bits. The parameter sets balance lattice security, ciphertext size, and decryption reliability. The generic

[3] `https://eprint.iacr.org/2019/1399`

[4] https://www.post-apocalyptic-crypto.org/timecop/

first-failure analysis in Section **??** uses at most 2^{80} classical decapsulation queries and a success probability of at least 1/2; the query limit alone does not impose a DFR bound. Table 2 gives DFR bounds and first-failure work lower bounds for the three compressed main parameter sets under the stated reference model. An independent correctness requirement must be assessed using its specified DFR definition and the actual compression distribution. Table 3 also includes the compact ZEN-light profile and parameter sets without compression.

Table 3: ZEN parameters

scheme	n	(D_g, D_f, D_s, D_e)	compression	PK	CT	`CoreSVP`
ZEN-light	512	$(B_1 + S_{1/16},\ S_{1/8},\ S_{3/16},\ B_1 + S_{1/8})$	$769 \to 128$	615	448	118
ZEN-128	512	$(B_1 + S_{1/8},\ B_1,\ B_1 + S_{3/32},\ B_2)$	$769 \to 256$	615	512	128
ZEN-256	1024	$(S_{3/16},\ S_{1/8},\ B_1,\ B_1)$	$769 \to 256$	1229	1024	257
ZEN-256-backup	1024	$(S_{3/16},\ S_{1/8},\ B_1,\ B_1)$	$769 \to 769$	1229	1229	257
ZEN-512	2048	$(S_{3/32},\ S_{3/32},\ S_{1/8},\ S_{1/8})$	$769 \to 256$	2458	2048	519
ZEN-512-backup	2048	$(S_{3/32},\ S_{3/32},\ S_{1/8},\ S_{1/8})$	$769 \to 769$	2458	2458	519

Table 3 lists the ZEN parameter sets and their public-key and ciphertext payload sizes in bytes. The `CoreSVP` values are lattice-estimator results; the corresponding `MATZOV` and other lattice estimates appear in Table 1. The modulus is shared across the parameter sets to maximize component reuse. ZEN-light provides a 448-byte ciphertext for constrained applications, but it is not the main 128-bit parameter set because its `CoreSVP` estimate is below 128 bits. The DFR and first-failure work bounds in Table 2 apply to ZEN-128, ZEN-256, and ZEN-512.

6.2 Protocol-level Usage

The size motivation for ZEN-light concerns the protocol-level object that carries an encapsulation output. The size of ZEN-light ciphertext is 448 bytes, which places it below several object-size boundaries that arise in constrained transports.

Several constrained transports expose protocol-object boundaries in this range. BLE ATT/GATT attribute values have a 512-octet maximum attribute value [8], so a 448-byte ciphertext can be represented as a single GATT attribute value. CoAP Block-wise transfer is designed to split representations into blocks rather than relying on IP fragmentation [9]; under a 512-byte block profile, a 448-byte ciphertext fits in one block with limited room for framing. Bluetooth BR/EDR L2CAP uses a 672-octet default MTU for the SDU size accepted on a channel [7], so a 448-byte ciphertext fits within that default object size without requiring MTU negotiation or higher-layer segmentation. BACnet MS/TP deployments also have APDU-size limits in this range; ASHRAE interpretation material discusses MS/TP and PTP APDU values around the 480-byte class, with further reductions possible when security overhead is included [4]. This should therefore be understood as an industrial-control motivation rather than as evidence of a single universal KEM ciphertext slot.

6.3 Implementation

6.3.1 Reference Implementation

The performance of the reference implementation was evaluated using the portable C implementation of ZEN. To assess the impact of different symmetric components on the overall running

time, we report two sets of timing results: one instantiated with SM3 and the other instantiated with FIPS202. All benchmarks were executed in a single-threaded setting on an Intel Core i9-11900K processor running at 3.5 GHz, with Turbo Boost and hyperthreading disabled to reduce measurement variance. The test platform was equipped with 32 GiB of memory and ran Ubuntu 22.04.2 LTS. The implementation was built using GNU Make 4.3 and compiled with GNU GCC 11.4.0 using the flags `-std=c99`, `-Wpedantic`, `-Wall`, `-Wextra`, and `-O2`. Each reported value represents the average number of CPU cycles over $100,000$ independent runs.

Table 4 presents the performance of the SM3-based reference implementation for all three supported security levels, namely ZEN-128, ZEN-256, and ZEN-512. Table 5 reports the corresponding timing results for the FIPS202-based reference implementation. Since the FIPS202-based instantiation is provided only for security levels up to 256 bits, no 512-bit performance result is reported for this variant.

Table 4: Performance of the SM3-Based Reference Implementation of ZEN

Scheme	Key Generation	Encapsulation	Decapsulation
ZEN-128	210661	123650	186536
ZEN-256	347851	172312	316429
ZEN-512	895633	460469	814405

Table 5: Performance of the FIPS202-Based Reference Implementation of ZEN

Scheme	Key Generation	Encapsulation	Decapsulation
ZEN-128	170208	68953	133739
ZEN-256	269512	123161	269729

6.3.2 Optimized Implementation on x86 with AVX2

The performance of the optimized implementation was evaluated using the x86 AVX2 implementation of ZEN. To assess the impact of different symmetric components on the overall running time, we report two sets of timing results. The first set corresponds to the SM3-based instantiation, where AVX2 optimizations are applied to all operations except the symmetric components, while SM3 itself remains implemented in standard C. Since we do not provide a dedicated AVX2-optimized implementation of SM3, the SM3-based results include the cost of the scalar SM3 component and are therefore less competitive than those of a fully vectorized symmetric instantiation. The second set corresponds to the FIPS202-based instantiation, for which an AVX2 implementation of the symmetric component is provided.

All benchmarks were executed in a single-threaded setting on an Intel Core i9-11900K processor running at 3.5 GHz, with Turbo Boost and hyperthreading disabled to reduce measurement variance. The test platform was equipped with 32 GiB of memory and ran Ubuntu 22.04.2 LTS. The implementation was built using GNU Make 4.3 and compiled with GNU GCC 11.4.0 using the flags `-O3`, `-march=x86-64`, `-mavx2`, `-mtune=native`, `-flto`, `-fomit-frame-pointer`, `-std=c99`, `-Wpedantic`, `-Wall`, and `-Wextra`. Each reported value represents the average number of CPU cycles over $100,000$ independent runs.

Table 6 presents the performance of the SM3-based AVX2 implementation for all three supported security levels, namely ZEN-128, ZEN-256, and ZEN-512. Table 7 reports the corresponding timing results for the FIPS202-based AVX2 implementation. Since the FIPS202-based instantiation is provided only for security levels up to 256 bits, no 512-bit performance result is reported for this variant.

Table 6: Performance of the SM3-Based AVX2 Implementation of ZEN with Plain-C SM3

Scheme	Key Generation	Encapsulation	Decapsulation
ZEN-128	87743	110557	110866
ZEN-256	178614	116795	125677
ZEN-512	537393	339675	346437

Table 7: Performance of the FIPS202-Based AVX2 Implementation of ZEN

Scheme	Key Generation	Encapsulation	Decapsulation
ZEN-128	22730	19487	24575
ZEN-256	38962	32149	47469

6.3.3 Optimized Implementation on ARM Cortex-M3

Our implementation of ZEN key encapsulation on Arm Cortex-M3 mainly involves two aspects. First, due to that `__uint128_t` is unsupported by compilers targeting 32-bit ARM architectures, we provide a 64×64 multiplication auxiliary function to support the 64-bits large integer multiplication required in public key unpacking. We decomposes 64-bits multiplication into multiple 32-bits multiplication and accumulation operations to adapt the large integer multiplication on this 32-bits processor. Secondly, to enhance the processing efficiency of ZEN on the Cortex-M3, we implement the assembly for Montgomery reduce, Montgomery multiplication, NTT, inverse NTT, base multiplication, and base inversion exploiting the Thumb-2 instruction set. We reorganize the computation steps and reusing intermediate values to optimize algorithmic execution flow, and make full use of the implementation level techniques, including improved register allocation, fewer loop-control ,etc , to reduce execution overhead.

We conducted the tests on the following set up: the targeted platform is Nucleo-F207ZG board which features a STM32F207ZG Cortex-M3 with 128 kB of SRAM and 1 MB of flash, and the core is clocked at 30 MHZ to avoid having flash wait states. The performance results of the implementation on ARM Cortex-M3 are presented in Table8.

Table 8: Performance of the Implementation on ARM Cortex-M3 (in cycles). Averaged over 100 iterations.

Scheme	Key generation	Encapsulation	Decapsulation
ZEN-128	625 217	575 064	814 230
ZEN-256	1 318 814	731 598	1 423 350
ZEN-512	3 694 895	1 963 327	3 858 533

6.3.4 Optimized Implementation on ARM Cortex-M4

The optimized ZEN implementation on ARM Cortex-M4 is based primarily on three categories of optimization techniques. First, we extensively employ the efficient Plantard arithmetic proposed in [25] to optimize polynomial arithmetic operations, including NTT, INTT, base multiplication, and base inversion. In particular, multiplications by twiddle factors in the NTT and INTT are accelerated using the efficient two-cycle Plantard multiplication by a constant from [25], which requires one fewer multiplication than Montgomery arithmetic. We further exploit the large input range supported by Plantard arithmetic to reduce the number of modular reductions in the NTT, INTT, base multiplication, and base inversion, thereby further improving the efficiency of these polynomial-arithmetic operations.

Secondly, in addition to polynomial arithmetic over $\mathcal{R}_{n,q}$ with $q = 769$, ZEN involves extensive operations in the binary polynomial ring $\mathcal{R}_n$. Modular multiplication over $\mathbb{F}_2$ is a major bottleneck that hinders the practical deployment of ZEN on low-end embedded devices. The reference implementation performs modular multiplication in $\mathbb{F}_2$ using exclusive-or operations, resulting in a time complexity of $O(n^2)$. This incurs significant overhead on 32-bit IoT platforms such as Cortex-M4 and becomes the dominant performance bottleneck in ZEN. To enable efficient deployment of ZEN on resource-constrained IoT devices, we optimize modular multiplication in $\mathbb{F}_2$ by adopting the radix-16 representation proposed in [12].

The radix-16 representation packs 8 coefficients of each element in $\mathbb{F}_2$ into a single 32-bit register:

$$a = a_0 + a_1 2^4 + a_2 2^8 + \cdots + a_7 2^{28}$$

With this representation, the multiplication of these eight coefficients in $\mathbb{F}_2$ can be performed using a conventional 32-bit multiplication instruction followed by a bitwise `and` operation. As a result, the time complexity is reduced from $O(n^2)$ to $O((n/8)^2)$. Moreover, the radix-16 representation can be combined with the divide-and-conquer strategy of Karatsuba multiplication to further improve the efficiency of modular multiplication. This optimization significantly accelerates fast inversion during key generation and simple decoding during public-key decryption.

Finally, as a side contribution, we also propose an efficient implementation of the SM3 hash function on ARM Cortex-M4. The proposed assembly implementation fully exploits the floating-point registers to cache intermediate results, thereby reducing costly memory accesses on Cortex-M4. In addition, we extensively leverage inline barrel-shifter operations to merge the rotation operations required by SM3 with other instructions, further improving the performance of the SM3 implementation.

Based on the optimization techniques described above, we present benchmark results for the ARM Cortex-M4 in Table 9. The benchmarks were conducted using our optimized ZEN implementation on the NUCLEO-L4R5ZI board, which features 2 MB of Flash and 640 KB of RAM. The experimental setup follows that of pqm4 [27]. All experiments were compiled with `arm-none-eabi-gcc` version 10.3.1 at the `-O3` optimization level. By combining the aforementioned optimizations, our optimized ZEN implementation outperforms the state-of-the-art ML-KEM implementation at the same security level from [25].

It is worth noting that our ZEN implementation uses SM3 as the pseudorandom number generator, whereas the ML-KEM implementation in [25] uses SHA3. Since SM3 is slower than

SHA3 when generating the same number of pseudorandom bytes, the relative performance advantage of ZEN over ML-KEM would be even larger if SHA3 were used as the generator in ZEN. These results highlight ZEN's efficiency in IoT scenarios and demonstrate its promise in the NGCC competition.

Table 9: Performance of the Additional Implementation on ARM Cortex-M4 (in cycles). Averaged over 100 iterations.

Scheme	Key generation	Encapsulation	Decapsulation
ZEN-128	350 655	268 493	422 335
ZEN-256	793 366	335 052	687 586
ZEN-512	2 416 438	964 852	1 864 525

6.3.5 Optimized Implementation on ARMv7

All architecture-specific optimizations are confined to the `ntt.c` module, while the remaining components of the implementation remain unchanged. To eliminate additional function call overhead, all NEON helper routines are implemented as `static inline` functions. The optimization introduces an eight-way vectorized Montgomery finite field multiplication routine, `neon_fqmul_8way`, together with a four-way parallel Montgomery reduction routine to accelerate modular multiplication while preserving bitwise equivalence with the scalar implementation through an unsigned mask-based truncation scheme. Furthermore, dedicated eight-way parallel butterfly kernels are developed for both the forward and inverse NTT, using NEON batch load/store instructions to process eight coefficients simultaneously and perform parallel addition, subtraction, and finite-field multiplication, thereby significantly reducing memory access and instruction overhead. Additional vectorization is applied to the final normalization stage of the inverse NTT, as well as to the Montgomery scaling and conditional Barrett correction procedures in `poly_ntt_mq` to eliminate scalar loops and further improve computational efficiency. The proposed optimization supports all three security parameter sets of the ZEN algorithm. For ZEN-128 ($N = 512$), NEON vectorization is applied to the upper computation stages while the lowest stage falls back to scalar processing for the remaining coefficients. For ZEN-256 ($N = 1024$) and ZEN-512 ($N = 2048$), all computation stages are fully vectorized without any scalar fallback.

Since `__uint128_t` is not supported by 32-bit ARM compilers, `div769_mul_shift` uses segmented 64-bit multiplication to implement the equivalent 128-bit multiply followed by a 58-bit right shift.

The tests are conducted on edge-intelligent convergence terminals, which are typical embedded devices used in real power-system scenarios, to evaluate the algorithm's practical runtime performance. The platform specifications are as follows: Operating system: Linux SCT230A with kernel version 3.10.108; CPU architecture: ARMv7l, quad-core, with a frequency ranging from 480 MHz to 1200 MHz; memory: 1.8 GB. The performance results of the additional implementation on ARMv7 are reported in Table 10.

Table 10: Performance of the Optimized Implementation on ARMv7 (in cycles). Averaged over 100000 iterations.

Scheme	Key generation	Encapsulation	Decapsulation
ZEN-128	499 627	397 045	525 577
ZEN-256	964 045	526 930	898 051
ZEN-512	2 776 204	1 418 549	2 607 733

Table 11: Stack usage of the Optimized Implementation on ARMv7 (in bytes).

Scheme	Key generation	Encapsulation	Decapsulation
ZEN-128	26 624	30 720	31 744
ZEN-256	35 840	29 696	31 744
ZEN-512	67 584	59 392	64 512

6.3.6 Optimized Implementation on ARMv8

The ARMv8 implementation is the performance-optimized software implementation for the key-encapsulation functionality of ZEN. It follows the NGCC KEM programming interface, including `kem_keygen`, `kem_keygen_derand`, `kem_enc`, and `kem_dec`. The auxiliary files supplied by the NGCC template, including `auxfunc.c`, `auxfunc.h`, `drng.c`, and `drng.h`, are kept unchanged.

The optimized ARMv8 code combines portable C with hand-optimized AArch64 assembly in GNU assembler syntax. The assembly components accelerate the performance-critical sampling routines, NTT-domain arithmetic, binary-ring operations, packed-bit fast inversion, and IND-CPA encryption/decryption subroutines. The implementation targets the ARMv8.2-A software environment with NEON, SVE, Crypto, SHA3, and DotProd support, and is compiled with `gcc` using `-O3 -march=armv8.2-a+crypto+sha3+dotprod+sve -flto -fomit-frame-pointer -std=c99 -Wpedantic -Wall -Wextra`.

The NTT-domain arithmetic follows the incomplete-NTT viewpoint used in high performance lattice implementations: after stopping the transform before the linear factors, the remaining base rings have the form $\mathbb{F}_q[x]/(x^m - \omega)$, and base multiplication or inversion is performed inside these small rings [1]. For the ZEN-128 base inversion over $\mathbb{F}_{769}[x]/(x^4 - \zeta)$, the assembly rewrites the degree-four inverse through a quadratic-extension norm. Writing $F = A + xB$, where $A = a_0 + a_2 y$, $B = a_1 + a_3 y$, $y = x^2$, and $y^2 = \zeta$, the inverse is computed as

$$F^{-1} = (A - xB)(A^2 - yB^2)^{-1}.$$

The remaining inverse is a degree-two inverse in $\mathbb{F}_{769}[y]/(y^2 - \zeta)$. This formulation exposes eight independent degree-four inversions to NEON lanes and reuses precomputed $\zeta, -\zeta$ pairs.

The modular arithmetic combines Montgomery multiplication with reciprocal Barrett-style reductions. Barrett reduction is a standard division-free modular reduction method [5]; signed and packed variants have been used to speed up NTT kernels in lattice implementations [38, 1]. In our ARMv8 assembly, twiddle multiplications use precomputed reciprocal constants together with vector high-half products and subtract-multiple steps, while lazy wide accumulations delay

reductions in the base multiplication kernels. These transformations preserve the same modulo-769 arithmetic as the C implementation while reducing the number of scalar reductions and memory accesses.

Following the NGCC ARM self-evaluation requirements, we evaluate the public KEM endpoints rather than internal kernels. Functionality is verified by KAT, by checking the interface length getters, and by checking that decapsulation recovers the encapsulated shared secret under both fixed-seed and DRNG-seeded tests. The measurements were conducted on a Huawei Cloud ARM instance running Ubuntu 24.04.4 LTS with Linux kernel 6.8.0-106-generic. The virtual CPU is reported by the platform as `2core, 4GB RAM of Huawei Hisilicon Kunpeng CPU @ 2.0GHz` on the AArch64 architecture. The benchmark was run as a single process pinned to CPU 0. Since the cloud instance does not expose hardware PMU cycle counters because it's running in QEMU, (`perf stat -e cycles,instructions` reports these events as unsupported) and the `/sys/devices/system/cpu/cpu0/cpufreq/` frequency files are not available, cycle counts are reported as estimates obtained by multiplying the measured wall-clock time by the reported virtual CPU frequency of 2.0 GHz.

The ARMv8 benchmark results appear in Table 12–Table 15. The functionality checks pass for all parameter sets.

Table 12: Performance of the Additional Implementation on ARMv8 (in µs).

Scheme	Key generation	Derandomized key generation	Encapsulation	Decapsulation	Correctness
ZEN-128	37.972	50.392	40.908	40.139	Pass
ZEN-256	76.060	81.470	41.766	41.836	Pass
ZEN-512	219.428	152.660	122.324	123.427	Pass

Table 13: Estimated cycle counts of the Additional Implementation on ARMv8.

Scheme	Key generation	Derandomized key generation	Encapsulation	Decapsulation
ZEN-128	75 944.63	100 784.91	81 815.52	80 278.23
ZEN-256	152 119.98	162 939.32	83 531.34	83 672.36
ZEN-512	438 856.68	305 320.25	244 647.35	246 853.41

Table 14: Throughput of the Additional Implementation on ARMv8 (operations per second).

Scheme	Key generation	Derandomized key generation	Encapsulation	Decapsulation
ZEN-128	26 334.98	19 844.24	24 445.24	24 913.36
ZEN-256	13 147.52	12 274.51	23 943.11	23 902.76
ZEN-512	4 557.30	6 550.50	8 175.03	8 101.97

Table 15: Memory consumption of the Additional Implementation on ARMv8 (in bytes).

Scheme	Static text/rodata	Static writable	Static total	Peak RSS after tests	Benchmark context
ZEN-128	45 768	85 504	131 272	1 871 872	2 590
ZEN-256	57 032	139 776	196 808	1 875 968	5 050
ZEN-512	114 536	82 272	196 808	1 875 968	9 972

6.3.7 Optimized Implementation on FPGA

For the hardware acceleration requirements of the ZEN parameter sets, we developed an FPGA-oriented dedicated implementation framework to provide unified acceleration for the three main flows: key generation, encapsulation, and decapsulation. This implementation is designed with synthesizability, verifiability, and scalability as its primary goals. At the top level, it maintains a consistent task-scheduling interface, while internally it adopts a modular structure centered around key computational components such as sampling, modular arithmetic, NTT/INTT, binary solving, packing, and unpacking. With this organization, different parameter sets can be deployed within a unified hardware framework while preserving consistency in the host control interface, data-path organization, and verification methodology.

Moreover, this implementation is not a straightforward flattening of the algorithmic flow into hardware. Instead, it performs targeted structural restructuring based on the synthesis and timing characteristics of FPGA platforms, particularly for long data paths, wide bit-selection networks, and high-fanout control paths. In this way, the design gradually evolves into an execution form that is more suitable for hardware implementation without altering the semantics of the main algorithmic flow, thereby providing a stable foundation for subsequent cycle optimization and engineering convergence.

The implementation uses a "software outer-layer control + hardware main computation core" architecture." In the `KEM` flow, the parts that are more control-intensive, involve relatively small amounts of data, and are not well suited to further hardware parallelization, such as random derivation, `pk` hashing, `kr` / fallback shared-secret generation, re-encryption result comparison, and final shared-secret selection, are all handled on the software side. In contrast, the main computation chains with the greatest cycle contribution, namely `PKE keygen / enc / dec`, continue to be executed on the FPGA hardware side. With this partitioning, hardware resources can be concentrated on dense computational paths such as polynomial arithmetic, sampling, and binary-domain solving, while the software side is responsible for outer-flow composition and a small amount of sequential control, thereby reducing the extra state and control overhead that would result from implementing the outer-layer `KEM` logic entirely in hardware.

From an architectural perspective, the top level does not adopt a scheme in which multiple arithmetic cores are issued simultaneously. Instead, it uses a unified buffer system and unified task control to serially schedule arithmetic cores such as `NTT`, `BASEMUL`, `BASEINV`, and `BINARY`. The actual parallelism is mainly retained within each core, while additional parallelism is exploited inside the algorithm wherever appropriate. This organization makes it possible to integrate different computation cores into a single hardware execution framework without reducing the effective parallelism of the main computation pipeline.

The division of labor between software and hardware is as follows:

- `KEM keygen`: hardware performs only `PKE keygen`, while software additionally handles `H(pk)`, random `z`, and final `KEM sk` assembly.

- `KEM enc`: software first performs message/random derivation, `H(pk)`, and `kr` generation, and hardware performs `PKE enc`.
- `KEM dec`: hardware first performs `PKE dec`, after which software handles `kr`, fallback `ss`, re-encryption comparison, and final `ss` selection.

The FPGA performance figures are based on synthesis and do not include on-board validation.

This benchmark reports only the cycle count of the hardware portion.

Table 16: Hardware latency of the FPGA RTL implementation (cycles).

Scheme	Key generation	Encapsulation	Decapsulation
ZEN-128	2 859	2 887	5 107
ZEN-256	7 197	3 936	7 626
ZEN-512	20 134	8 621	17 959

7 Feature

Innovation ZEN fully exploits the available algebraic structure while maintaining careful security control. Its delicate yet succinct construction makes it easy to understand in theory and straightforward to implement in practice.

Compactness ZEN exploits the redundancy of the message space to reduce noise and mitigate decryption failure, thereby substantially decreasing parameter sizes. ZEN-light has a 615-byte public key and a 448-byte ciphertext, which helps avoid packet fragmentation in protocols with small packet-size limits. At the same security level, ZEN also offers size advantages over mainstream lattice-based encryption schemes.

Efficiency ZEN mainly consists of polynomial multiplication, polynomial inversion, and decoding via an empirical key-exchange procedure. Polynomial multiplication and inversion over $\mathbb{Z}_q$ rely on the NTT for high efficiency, while polynomial inversion over $\mathbb{Z}_2$ is accelerated using a Hensel-lifting-style technique. The decoding procedure is also fast, since it consists primarily of XOR operations.

Security ZEN is based on standard and well-studied lattice assumptions, namely decisional NTRU and decisional Ring-LWE over power-of-two cyclotomic rings. The KEM layer follows an implicit-rejection Fujisaki–Okamoto transform, and the decryption-failure analysis is formulated through a group-level upper bound. For practical security evaluation, ZEN uses conservative estimator settings, including the `CoreSVP` model and security estimates that ignore the additional noise introduced by compression. These choices make the reported security margins deliberately cautious while keeping the assumptions and evaluation methodology aligned with established lattice-based cryptography practice.

References

[1] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-m4. Cryptology ePrint Archive, Paper 2022/112, 2022.

[2] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate All the {LWE, NTRU} Schemes! In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 351–367, Cham, 2018. Springer International Publishing.

[3] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[4] ASHRAE. Interpretation IC 135-2008-10 of ANSI/ASHRAE Standard 135-2008, 2010.

[5] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[6] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: Reducing Attack Surface at Low Cost. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, pages 235–260, Cham, 2018. Springer International Publishing.

[7] Bluetooth Special Interest Group. Bluetooth Core Specification Version 5.4, Vol. 3, Part A: Logical Link Control and Adaptation Protocol Specification, 2023.

[8] Bluetooth Special Interest Group. Bluetooth Core Specification Version 5.4, Vol. 3, Part F: Attribute Protocol (ATT), 2023.

[9] Carsten Bormann and Zach Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, 2016.

[10] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, 2018.

[11] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas H"ulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU: Algorithm Specifications and Supporting Documentation. Brown University and Onboard Security Company, Wilmington, USA, 2019.

[12] Ming-Shing Chen and Tung Chou. Classic mceliece on the ARM cortex-m4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):125–148, 2021.

[13] Léo Ducas and Ludo N. Pulles. Does the dual-sieve attack on learning with errors even work? In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 37–69, Cham, 2023. Springer Nature Switzerland.

[14] Léo Ducas and Wessel van Woerden. NTRU Fatigue: How Stretched is Overstretched? In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 3–32, Cham, 2021. Springer International Publishing.

[15] Julien Duman, Kathrin Hövelmanns, Eike Kiltz, Vadim Lyubashevsky, Gregor Seiler, and Dominique Unruh. A Thorough Treatment of Highly-Efficient NTRU Instantiations. In *Public-Key Cryptography PKC 2023: 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 710, 2023, Proceedings, Part I*, page 6594, Berlin, Heidelberg, 2023. Springer-Verlag.

[16] Pierre-Alain Fouque, Paul Kirchner, Thomas Pornin, and Yang Yu. BAT: Small and fast KEM over NTRU lattices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):240–265, 2022.

[17] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. Cryptology ePrint Archive, Report 2000/061, 2000.

[18] Lewis Glabush, Kathrin Hövelmanns, and Douglas Stebila. On The Multi-target Security of Post-Quantum Key Encapsulation Mechanisms. *IACR Communications in Cryptology*, 3(1), 2026.

[19] Lewis Glabush, Patrick Longa, Michael Naehrig, Chris Peikert, Douglas Stebila, and Fernando Virdia. FrodoKEM: A CCA-secure learning with errors key encapsulation mechanism. *IACR Communications in Cryptology*, 2(3), 2025.

[20] Qian Guo and Thomas Johansson. Faster Dual Lattice Attacks for Solving LWE with Applications to CRYSTALS. In *Advances in Cryptology ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 610, 2021, Proceedings, Part IV*, page 3362, Berlin, Heidelberg, 2021. Springer-Verlag.

[21] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[22] Jeffrey Hoffstein and Joseph Silverman. Optimizations for NTRU. In Kazimierz Alster, Jerzy Urbanowicz, and Hugh C. Williams, editors, *Public-Key Cryptography and Computational Number Theory*, pages 77–88, Berlin, New York, 2001. De Gruyter.

[23] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A Modular Analysis of the Fujisaki-Okamoto Transformation. In *TCC (1)*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, 2017.

[24] Jianhua Hou and Haodong Jiang. Careful with the ring: Enhanced hybrid decoding attacks against module/ring-LWE. Cryptology ePrint Archive, Paper 2026/366, 2026.

[25] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):614–636, 2022.

[26] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. IND-CCA-Secure Key Encapsulation Mechanism in the Quantum Random Oracle Model, Revisited. In *CRYPTO (3)*, volume 10993 of *Lecture Notes in Computer Science*, pages 96–125. Springer, 2018.

[27] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.

[28] Yijian Liu, Yu Zhang, Xianhui Lu, Yao Cheng, and Yongjian Yin. DAWN: Smaller and Faster NTRU Encryption via Double Encoding. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025*, pages 396–427, Singapore, 2026. Springer Nature Singapore.

[29] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer Berlin Heidelberg, Germany.

[30] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly Fast NTRU Using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180201, May 2019.

[31] MATZOV. Report on the Security of LWE: Improved Dual Lattice Attack. 2022.

[32] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication FIPS 203, National Institute of Standards and Technology, 2024.

[33] Tabitha Ogilvie. On the concrete hardness gap between MLWE and LWE. Cryptology ePrint Archive, Paper 2026/279, 2026.

[34] Thales B. Paiva, Marcos A. Simplício Jr., Syed Mahbub Hafiz, Bahattin Yildiz, Eduardo Lopes Cominetti, and Henrique S. Ogawa. Tailorable codes for lattice-based KEMs with applications to compact ML-KEM instantiations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):139–163, 2025.

[35] Alice Pellet-Mary and Damien Stehlé. On the Hardness of the NTRU Problem. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 3–35, Cham, 2021. Springer International Publishing.

[36] Amaury Pouly and Yixin Shen. Provable dual attacks on learning with errors. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 256–285, Cham, 2024. Springer Nature Switzerland.

[37] Hongyuan Qu and Guangwu Xu. On the provable dual attack for lwe by modulus switching. In *Advances in Cryptology ASIACRYPT 2025: 31st International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, VIC, Australia, December 812, 2025, Proceedings, Part III*, page 3464, Berlin, Heidelberg, 2025. Springer-Verlag.

[38] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018.

[39] Damien Stehlé and Ron Steinfeld. Making NTRU as Secure as Worst-Case Problems over Ideal Lattices. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 27–47, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[40] Yang Wang and Mingqiang Wang. Provably Secure NTRUEncrypt over Any Cyclotomic Field. In *ACM Symposium on Applied Computing*, 2018.

[41] Jiang Zhang, Dengguo Feng, and Di Yan. NEV: Faster and smaller NTRU encryption using vector decoding. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part VII*, volume 14444 of *Lecture Notes in Computer Science*, pages 157–189, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore.