Shorter Messages and Faster Post-Quantum Encryption with Round5 on Cortex M

Markku-Juhani O. Saarinen[1], Sauvik Bhattacharya[2], Oscar Garcia-Morchon[2], Ronald Rietman[2], Ludo Tolhuizen[2], and Zhenfei Zhang[3]

[1] PQShield Ltd., United Kingdom. Email: mjos@pqshield.com
[2] Philips, Netherlands. Email: first.lastname@philips.com
[3] OnBoard Security, USA. Email: zzhang@onboardsecurity.com

Abstract. Round5 is a Public Key Encryption and Key Encapsulation Mechanism (KEM) based on General Learning with Rounding (GLWR), a lattice problem. We argue that the ring variant of GLWR is better suited for embedded targets than the more common RLWE (Ring Learning With Errors) due to significantly shorter keys and messages. Round5 incorporates GLWR with error correction, building on design features from NIST Post-Quantum Standardization candidates Round2 and Hila5. The proposal avoids Number Theoretic Transforms (NTT), allowing more flexibility in parameter selection and making it simpler to implement. We discuss implementation techniques of Round5 ring variants and compare them to other NIST PQC candidates on lightweight Cortex M4 platform. We show that Round5 offers not only the shortest key and ciphertext sizes among Lattice-based candidates, but also has leading performance and implementation size characteristics.

Keywords: Post-Quantum Cryptography, Lattice Cryptography, GLWR, Embedded Implementation, Cortex M4

1 Introduction

There is well-founded speculation that the estimated time required for development of quantum computers capable of breaking RSA and Elliptic Curve Cryptography (ECC) [25,30] is shorter than the long term confidentiality requirements of some current highly sensitive communications and data. Such risk analysis prompted the National Security Agency (NSA) to revise its cryptographic algorithm recommendations in 2015 and to announce a "transition period" until quantum resistant replacement algorithms can be fielded [6,23].

The algorithm identification and standardization task fell largely to National Institute of Standards and Technology (NIST), who specified evaluation criteria and organized a public call for Post-Quantum Cryptography (PQC) algorithms in 2016 [20,21]. A total of 69 public key encryption, key encapsulation, and digital signature algorithm submissions were made by the November 2017 deadline [22].

The new proposals rely on a wide variety of quantum-resistant hard problems from areas such as lattices, coding theory, isogenies of supersingular curves, and

multivariate equations. A set of selected PQC algorithms is expected to eventually fulfill all of the tasks that have up to now been assigned to classically secure (RSA and ECC) public key algorithm standards. This includes cryptography in lightweight embedded applications and smart cards.

2 Round5

Round5 is an amalgam of two lattice-based first-round candidates in the NIST Post-Quantum cryptography project, Round2 [14] and Hila5 [29]. Like its two parent proposals, Round5 can be used for both public key encryption and key encapsulation, and it inherits the use of a rounding problem from Round2 (GLWR, Section 2.1) and error correction from Hila5 (XEf, Section 3.1).

The use of a rounding problem together with error correction lends Round5 unique bandwidth efficiency properties. A full description of Round5, its design, classical and quantum security analysis, and parameter selection can be found in [5]. Details of that analysis are outside the scope of this work but we note that the new parameter selection addresses the potential issues regarding classical attack bounds in the original Round2 submission.

2.1 Generalized Learning With Rounding

There is a relatively large set of interrelated hard problems used in lattice cryptography. One of the most common ones is Learning With Errors (LWE), which has a security reduction to worst-case quantum hardness of shortest vector problems GapSVP and SIVP [26,27]. Learning With Rounding (LWR) was introduced in [3], where it was shown to have a security reduction from LWE. Round2 utilizes a version called General Learning With Rounding (GLWR).

A key feature of Round5 is the use of *rounding* in the form of a lossy compression function, Round. It maps $x \in \mathbb{Z}_a$ to $\mathbb{Z}_b$ with rounding constant h:

$$\mathsf{Round}_{a\to b}(x,h) = \left\lfloor \frac{b}{a} \cdot x + h \right\rfloor \bmod b. \tag{1}$$

This is equivalent to rounding of $^{bx}/_{a}$ to closest integer when $h = {}^{1}/_{2}$. Each coefficient is operated on separately when Round or modular reduction ("mod") is applied to polynomials, vectors, or matrices.

Definition 1 (General LWR (GLWR)). *Let d, n, p, q be positive integers such that $q \geq p \geq 2$, and $n \in \{ 1, d \}$. Let $\mathcal{R}_{n,q}$ be a polynomial ring, and let D_s be a probability distribution on $\mathcal{R}_n^{d/n}$.*

- *The search version of the GLWR problem $sGLWR_{d,n,m,q,p}(D_s)$ is as follows: given m samples of the form $(\mathbf{a}_i, b_i = \mathsf{Round}_{q\to p}(\mathbf{a}_i^T\mathbf{s} \bmod q, {}^{1}/_{2})$ with $\mathbf{a}_i \in \mathcal{R}_{n,q}^{d/n}$ and a fixed $\mathbf{s} \leftarrow D_s$, recover $\mathbf{s}$.*
- *The decision version of the GLWR problem $dGLWR_{d,n,m,q,p}(D_s)$ is to distinguish between the uniform distribution on $\mathcal{R}_{n,q}^{d/n} \times \mathcal{R}_{n,p}$ and the distribution $(\mathbf{a}_i, b_i = \mathsf{Round}_{q\to p}(\mathbf{a}_i^T\mathbf{s}_i \bmod q, {}^{1}/_{2}))$ with $\mathbf{a}_i \leftarrow \mathcal{R}_{n,q}^{d/n}$ and a fixed $s \leftarrow D_s$.*

2.2 Highly Flexible Parameters: Embedded Use Case

The $n = 1$ case of GLWR corresponds to the original LWR problem of [3]. In this work we restrict ourselves to the $n = d$ case, which corresponds to the Ring-LWR (RLWR) problem and offers shorter public keys and ciphertext messages. See [5] and Table 4 for a full list of parameter sets.

Round5 has both chosen ciphertext (CCA) and chosen plaintext (CPA) secure versions. The CPA versions are faster and are configured to have smaller keys at the price of a slightly higher failure rate, making them better suited for ephemeral key establishment. On the other hand, parameter selection leading to a negligible error rate and the added security of CCA Fujisaki-Okamoto Transform [13,15] is needed in public key encryption applications, where messages and public keys have long lifetimes. Therefore the CCA variant is referred to as "Round5.PKE", while the CPA version is called "Round5.KEM". They both internally rely on the same building lock, an IND-CPA encryption scheme. Since both key establishment and public key encryption use cases are relevant to embedded applications, we consider them both.

In addition to the LWR / RLWR and CCA / CPA distinctions, Round5 defines parameter sets for each NIST encryption security category NIST1, NIST3, and NIST5. These correspond to the security level of AES with 128, 192, and 256 - bit key length, respectively, against a quantum or classical adversary [21]. This leads to a total of $2 \times 2 \times 3 = 12$ different Round5 variants.

However all applications clearly don't need to implement all variants. We adopt the strategy taken in NSA's Commercial National Security Algorithm (CNSA) suite [23] which standardizes on a single set of parameters and algorithms at 192-bit (classical) security level. This facilitates interoperability and parameter-specific implementation optimizations, leading to smaller implementation footprint. CNSA is approved up to TOP SECRET in United States.

Therefore this work focuses on two variants with designators `R5ND_3KEM` and `R5ND_3PKE`. One can read the designators aloud as "Round 5" (`R5`), "ring variant" (`ND` for $n = d$), "post-quantum security category 3" (`3`), "CPA security for ephemeral keys" (`KEM`) or "CCA security for public key encryption" (`PKE`).

2.3 High-Level Algorithm Overview

Table 1 summarizes the internal and external parameters of our implementation.

Round5 uses two polynomial rings[4]; $x^{n+1}-1$, with $n+1$ prime, and its subring $\Phi_{n+1} = (x^{n+1} - 1)/(x - 1) = x^n + \cdots + x + 1$. We observe that $x^n \bmod \Phi_{n+1} = -\sum_0^{n-1} x^i$. Therefore one can utilize a trick for reducing modulo Φ_{n+1}, first reducing a result modulo cyclic $x^{n+1}-1$ where $x^{i+n+1} \equiv x^i$, and then subtracting the x^n coefficient from the rest of coefficients (and itself).

The small-norm secrets have special structure: sparse ternary polynomial set $D \subset \{-1, 0, 1\}^n$ has $\frac{h}{2}$ coefficients set to +1, $\frac{h}{2}$ coefficients set to -1, and $n - h$ coefficients being zero.

[4] Originally only one ring was used. As pointed out by Mike Hamburg, use of two rings yields better error analysis, and works much better with error correction.

Table 1. Internal parameters and external attributes for the R5ND3 variants of Round5 discussed in this paper. The security estimates are made with very conservative assumptions and correspond to NIST3 security level. There are many more variants available – please see Table 4 for a full list and [5] for the paremeter selection methodology used. **NOTE: Parameters are being revised due to ring change.**

Parameter	R5ND_3KEM	R5ND_3PKE
Dimension	$n = 756$	$n = 786$
Degree ($n = d$ for ring variants)	$d = 756$	$d = 786$
Nonzero elements in ternary secrets	$h = 242$	$h = 204$
Large (main) modulus	$q = 2^{15}$	$q = 2^{16}$
Rounding modulus	$p = 2^8$	$p = 2^8$
Compression modulus	$t = 2^4$	$t = 2^6$
Encrypted secret size (bits)	$\|K\| = 192$	$\|K\| = 192$
Error correction code size (bits)	$l = 103$	$l = 103$
Transmitted secret (bits $\mu = \|K\| + l$)	$\mu = 295$	$\mu = 295$
Random bit flips corrected (by XEf)	$f = 3$	$f = 3$
Public key size (bytes)	780	810
Secret key size (bytes)	24	858
Ciphertext expansion (bytes)	904	1032
Shared secret size (bytes)	24	24
Quantum security	2^{176}	2^{181}
Classical security	2^{193}	2^{193}
Decryption failure rate	2^{-75}	2^{-128}

Ignoring a lot of detail, the basic key generation procedure KeyGenCPA() is given by Algorithm 1 while Algorithms 2 and 3 describe the basic encryption and decryption operations EncryptCPA() and DecryptCPA(), respectively. The function Sample_μ takes μ lowest-order coefficients of input. Note that the rounding constant for Round is not always $1/2$ for non-ring variants.

To see why the algorithm works, note that the shared secrets in Algorithms 2 and 3 satisfy approximately $\mathsf{t} \approx \mathsf{t}' \approx \mathsf{a}*\mathsf{s}*\mathsf{r}$. Even though the two multiplications are in a different rings, the second ring is a subring of the first. Since s and r are "balanced", their coefficients sum to zero and they are divisible by $(x-1)$. High bits of t are used as a "one time pad" to transport the message payload.

CPA-KEM. The chosen-plaintext secure (IND-CPA) key encapsulation mode is constructed from EncryptCPA() and DecryptCPA() in straightforward fashion by randomizing seed ρ and composing the message $\mathbf{m} \in \{0,1\}^\mu$ from random key $\mathbf{k} \xleftarrow{\$} \{0,1\}^{|K|}$ and an error correction code (Section 3.1). Both parties compute the shared secret as $\mathtt{ss} = h(\mathbf{k}, \mathtt{ct})$ (after error correction in decapsulation.)

CCA-KEM. The CPA scheme is transformed into a chosen ciphertext (IND-CCA2) secure one (in R5ND_3PKE) using the Fujisaki-Okamoto Transform [13,15]:

Algorithm 1 $\mathsf{KeyGenCPA}(\sigma, \gamma)$: Key generation for CPA case.

Input: Random seeds σ, γ.

1: $\mathbf{a} \xleftarrow{\$_\sigma} \mathbb{Z}_q^n$	*Uniform polynomial, seed* σ.
2: $\mathbf{s} \xleftarrow{\$_\gamma} D$	*Sparse ternary polynomial, seed* γ.
3: $\mathbf{b} \leftarrow \mathsf{Round}_{q\to p}(\mathbf{a} * \mathbf{s} \mod \Phi_{n+1}, 1/2)$	*Compress product to range* $0 \le b_i < p$.
4: $\mathtt{sk} = \mathbf{s}$	$\mathbf{s}$*: Random seed* γ *is sufficient.*
5: $\mathtt{pk} = (\mathbf{a}, \mathbf{b})$	$\mathbf{a}$*: Random seed* σ*,* $\mathbf{b}$*:* $n \log_2 p$ *bits.*

Output: Public key $\mathtt{pk} = (\mathbf{a}, \mathbf{b})$ and secret key $\mathtt{sk} = \mathbf{s}$.

Algorithm 2 $\mathsf{EncryptCPA}(\mathbf{m}, \mathtt{pk}, \rho)$: Public key encryption (CPA).

Input: Message $\mathbf{m} = \{0,1\}^m$, public key $\mathtt{pk} = (\mathbf{a}, \mathbf{b})$, random seed ρ.

1: $\mathbf{r} \xleftarrow{\$_\rho} D$	*Sparse ternary polynomial, seed* ρ
2: $\mathbf{u} \leftarrow \mathsf{Round}_{q\to p}(\mathbf{a} * \mathbf{r} \mod \Phi_{n+1}, 1/2)$	*Compress product to range* $0 \le u_i < p$.
3: $\mathbf{t} \leftarrow \mathsf{Sample}_\mu(\mathbf{b} * \mathbf{r} \mod x^{n+1} - 1)$	*Noisy shared secret, truncate to* $\mathbb{Z}_p^\mu$.
4: $\mathbf{v} \leftarrow \mathsf{Round}_{p\to t}(\mathbf{t} + \frac{p}{2}\mathbf{m}, 1/2)$	*Add message + error correction* $\mathbf{m} \in \mathbb{Z}_2^\mu$.
5: $\mathtt{ct} = (\mathbf{u}, \mathbf{v})$	$\mathbf{u}$*:* $n \log_2 p$ *bits,* $\mathbf{v}$*:* $\mu \log_2 t$ *bits.*

Output: Ciphertext $\mathtt{ct} = (\mathbf{u}, \mathbf{v})$.

– **Key generation** requires storing secret coins $\mathbf{z} \xleftarrow{\$} \{0,1\}^{|K|}$ and the public key from $\mathsf{KeyGenCPA}$ with the secret key: $\mathtt{CCAsk} = (\mathtt{sk}, \mathbf{z}, \mathtt{pk})$.
– **Encapsulation.** We hash $\mathbf{m} \in \{0,1\}^\mu$ consisting of a random message and error correction with the public key to create a triplet of $|K|$-bit quantities $(\mathbf{l}, \mathbf{g}, \rho) = h(\mathbf{m}, \mathtt{pk})$. Then compute $\mathbf{c} = (\mathsf{EncryptCPA}(\mathbf{m}, \mathtt{pk}, \rho)$ and set ciphertext as $\mathtt{ct} = (\mathbf{c}, \mathbf{g})$. The shared secret is $\mathtt{ss} = h(\mathbf{l}, \mathtt{ct})$.
– **Decapsulation** computes $\mathbf{m}' = \mathsf{DecryptCPA}(\mathbf{c}, \mathtt{sk})$ from the first part of ciphertext and uses that to create its version of triplet $(\mathbf{l}', \mathbf{g}', \rho') = h(\mathbf{m}', \mathtt{pk})$. This is then used in simulated encryption $\mathbf{c}' = \mathsf{EncryptCPA}(\mathbf{m}', \mathtt{pk}, \rho')$. If there is a match $\mathtt{ct} = (\mathbf{c}', \mathbf{g}')$, we set $\mathtt{ss} = h(\mathbf{l}', \mathtt{ct})$. In case of mismatch $\mathtt{ct} \neq (\mathbf{c}', \mathbf{g}')$ we use our stored coins $\mathbf{z}$ for deterministic output $\mathtt{ss} = h(\mathbf{z}, \mathtt{ct})$.

KEMs and Public Key Encryption. While `R5ND_3KEM` (CPA) is sufficient for purely ephemeral key establishment, we suggest `R5ND_3PKE` (CCA) for public key encryption [7]. One of course needs to further define a Data Encapsulation Mechanism (DEM) in order to transmit actual messages rather than just keys.

Algorithm 3 $\mathsf{DecryptCPA}(\mathtt{ct}, \mathtt{sk})$: Decryption (CPA).

Input: Ciphertext $\mathtt{ct} = (\mathbf{u}, \mathbf{v})$, secret key $\mathtt{sk} = \mathbf{s}$.

1: $\mathbf{t}' \leftarrow \mathsf{Sample}_\mu(\mathbf{u} * \mathbf{s} \mod x^{n+1} - 1)$	*Noisy shared secret, truncate to* $\mathbb{Z}_p^\mu$.
2: $\mathbf{m} \leftarrow \mathsf{Round}_{p\to 1}(\frac{p}{t}\mathbf{v} - \mathbf{t}', 1/2)$	*Remove noise, recover message* $\mathbf{m} \in \mathbb{Z}_2^\mu$.

Output: Plaintext $\mathtt{pt} = \mathbf{m}$.

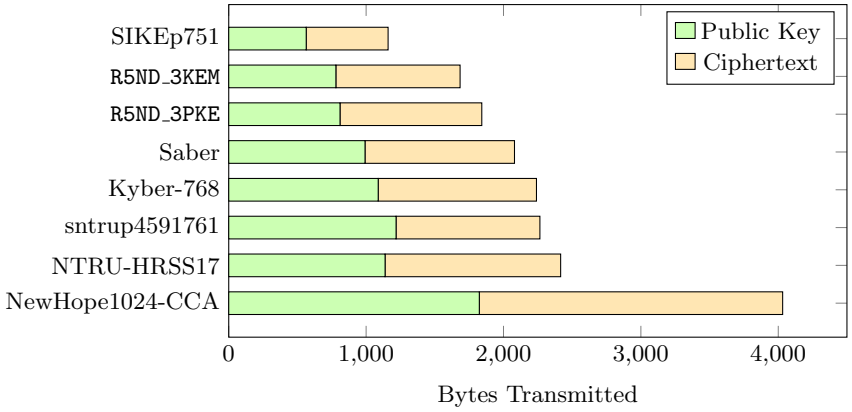

Fig. 1. Bandwidth usage in key establishment. In addition to other relatively bandwidth-efficient lattice schemes, we include SIKEp751, which is the only candidate with shorter messages at level 3.

3 Implementation Tweaks and Optimizations

The operation of Round5 in a ring is analogous to "LP11" [19] encryption, but using rounding instead of synthetic random error. Our lightweight implementation in particular shares some similarity with "half-truncated" lightweight Ring-LWE scheme TRUNC8 [28], but uses a sparse ternary vector instead of a binary secret. Here we highlight some key factors the embedded implementation to be faster and more compact than the reference implementation (and Round2).

Fixed parameters. The original implementation of Round2 was an extremely flexible "all-rounder" – the source code of numerous variants differs only in the `api.h` file. Although that implementation showed that it is feasible to have good performance for multiple parameter sets, fixing parameters to a small set is desirable especially for embedded bare metal targets, as that lets us eliminate dynamic (heap) memory allocation. Fixed parameters also allow us to optimize things such as array indexing and to remove redundant loops and options. This alone led to roughly 3× overall speedup of `R5ND` in comparison to an implementation of Round5 capable of handling all parameter sets.

Simplifications. There are a number of practical simplifications related to our specific parameter choices. Since p and q are powers of two, there is a lot of masking by $p-1$ and $q-1$. However much of this is unnecessary since carry bits do not flow from higher bits towards lower bits in addition and subtraction. Therefore all intermediate values can be kept at full word length. Most of the arithmetic operates internally on 16-bit words, well suited for lightweight targets.

SHAKE-256. We use SHAKE-256 [11] consistently for hashing and random byte sequence generation. Round2 used SHA3-512 for "short-output hashing", usually truncating the result to 32 bytes. SHA3-512 with its 1024-bit internal state and slow speed is clearly an overkill. Round2 furthermore specified a "DRBG" based on AES-256 [10] in counter mode [9]. SHAKE-256 is designed as a extendable output function (XOF) and takes over the functions of DRBG.

There were instances of double hashing within the algorithm, such as hashing input to get a fixed-length DRBG seed – which is of course unnecessary in case of an arbitrary-input XOF. Another case was the three-output G function which was previously implemented with three iterations of hashing rather than cutting a longer XOF output into pieces.

Faster generation of sparse ternary vectors. We use a rejection sampling method rather than the sorting method originally used in Round2. This faster method allows us to store a random seed instead of a full ternary vector as the secret key. See Section 3.3 and Algorithm 4 for more details. The original method was chosen to have constant time execution (even though it didn't always have that in practice). We note that even though rejection sampling has variable execution time, it does not leak secrets iff the distribution is constant, the original secret values are statistically independent, and a non-rejected result itself does not cause a timing variation (e.g. via memory accesses).

3.1 Error Correcting Code XEf

A 3-error correcting block code is used to decrease the failure rate. The code is built using the same strategy as codes used by TRUNC8 [28] (2-bit correction) and HILA5 [29] (5-bit correction).

Our linear parity code consists of $2f = 6$ "registers" R_i of size $|R_i| = l_i$. We view the payload block m as a binary polynomial $m_{|K|-1}x^{|K|-1} + \cdots + m_1 x + m_0$ of length equivalent to shared secret K. Registers are defined via cyclic reduction

$$R_i = m \bmod x^{l_i} - 1, \tag{2}$$

or equivalently by

$$r_{(i,j)} = \sum_{k \equiv j \bmod l_i} m_k \tag{3}$$

where $r_{(i,j)}$ is bit j of register R_i. A transmitted message consists of the payload m concatenated with register set r (a total of $|K| + \sum l_i$ bits).

Upon receiving a message $(m' \mid r)$ one computes code r' corresponding to m' and compares it to the received code r – that may also have errors. Errors are in coefficients m'_j where there is parity disagreements $r_{(i,j \mod l_i)} \neq r'_{(i,j \mod l_i)}$ for multitude of registers R_i. We use a majority rule and flip bit m'_j if

$$\sum_{i=1}^{2f} \left(\left(r_{(i,j \mod l_i)} - r'_{(i,j \mod l_i)} \right) \bmod 2 \right) \geq f + 1 \tag{4}$$

where the sum is taken as the number of disagreeing register parity bits at j.

It is easy to show that if all length pairs satisfy $\text{lcm}(l_i, l_j) \geq |K|$ when $i \neq j$ then this code always always corrects at least f errors. Typically one chooses coprime lengths $l_1 < l_2 < \cdots < l_{2f}$ so that $l_1 l_2 \geq |K|$.

Our variants have $f = 3$ and $(l_1, l_2, \cdots, l_6) = (13, 15, 16, 17, 19, 23)$. The code adds $\sum_i l_i = l = 103$ bits to the message, bringing the total to $192 + 103 = 295$ bits. We have verified that our implementation always fixes 3 bit flips anywhere in the 295-bit block and 4 bit flips with probability $44785504/309177995 \approx 14.5\%$. Its main advantage over other error-correcting codes is that it can be implemented without table look-ups and conditional cases and it is therefore resistant to timing attacks. See Tables 1 and 3 for overall failure rate estimates.

3.2 Arithmetic of Sparse ternary polynomials

Unlike many other fast lattice-based schemes, our `R5ND3` variants do not use the (Nussbaumer) Number Theoretic Transform (NTT) for its ring arithmetic [24]. This allows more flexibility for selection of n and greater variance in implementation techniques leading to substantial reduction in implementation footprint.

Multiplication of a ring element with $\{-1, 0, +1\}$ coefficients requires only additions and subtractions. Furthermore the use of power-of-2 moduli q, p, t means that no modular reduction is required. This greatly simplifies implementation, especially on hardware targets, but also on microcontrollers without a multiplier (where performance gains are likely to be more significant than on Cortex M).

Implementation of sparse ternary multiplication required special attention as that is the workhorse of Round2 and a large portion of its execution time is spent performing this operation. Clearly its complexity is $O(hn)$ but even though this is asymptotically worse than $O(n \log n)$ of NTT, our findings indicate that it is significantly better in practice with the parameters of `R5ND_3KEM` and `R5ND_3PKE`.

The lowest level loops in multiplication are simple vector additions and subtractions. Since there is an equivalent number $(h/2)$ of +1 and -1 terms in the ternary polynomials, the computation is organized in a way that allows an addition and an subtraction to be paired in a each loop.

Similar techniques are highly effective on SIMD targets such as AVX2 as well, but require special cache attack countermeasures. Cache attacks are not a concern with Cortex M SoCs (since all memory is internal to the chip and there is no RAM cache [5]) but we note that our cache-resistant portable implementation runs at about half of the speed of the normal version.

3.3 Sparse ternary vector generation

Algorithm 4 describes our deterministic method for creating sparse ternary vectors of weight h. It uses rejection sampling to obtain uniformly random index $0 \leq t < n$. This is clearly not a constant time operation – however we can see that a rejection sampler does not leak information about t since bytes in $\mathbf{z}$ are

[5] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0321a`

Algorithm 4 SparseTernary(s): A ternary vector with weight h from seed s.

Input: Seed value s, dimension and degree n, scaling factor $k = \lfloor 2^{16}/n \rfloor$.

```
1: z ← SHAKE256(s)                    Absorb the seed s into Keccak state.
2: v ← 0^n                            Initialize as zero.
3: for i = 0, 1, ... h − 1 do
4:    repeat
5:       repeat
6:          t' ← two bytes from z      z represents the (endless) output of XOF.
7:       until t' < kn                 Rejection step with the unscaled value.
8:       t ← ⌊t'/k⌋                    Remove the integer scaling factor k.
9:    until v_t = 0                    Another rejection. Vector is sparse.
10:   v_t ← (−1)^i                     Alternating +1, −1, +1, ··· .
11: end for
```

Output: A vector $\mathbf{v}$ which has $\frac{h}{2}$ elements set to +1, $\frac{h}{2}$ set to -1 and $n - h$ zeros.

statistically independent. This has caused some false positives when automated tools are used to detect timing leaks.

Even though Algorithm 4 produces correct results, in practice the vector $\mathbf{v}$ is only used to store only an "occupancy table" of free slots, while the actual indices t are stored in two lists of $+1$ and -1 coefficient offsets (which correspond to coefficient of degree $n+1-t$ in ring polynomial representation). These lists of length $h/2$ are used in multiplication rather than scanning $\mathbf{v}$. There is no reason to sort the lists – a randomized pattern may even work as a free cache attack countermeasure on targets where that may be a problem.

The use of scaling factor k greatly lowers the rejection rate of the inner sampler. With `R5ND_3KEM` we have $n = 700$, $k = \lfloor 2^{16}/700 \rfloor = 93$, leading to rejection rate of just $1 - kn/2^{16} = 0.00665$, while `R5ND_3PKE` has $n = 756$, $k = 86$ and rejection rate of 0.00793.

There is a secondary rejection when finding non-empty slots in the v vector, which might make an algorithm of this sort run in essentially quadratic time if h is close to n. However in our case the density is bound by $h/n \leq 28\%$.

A simple implementation of secondary rejection should be timing attack resistant on the Cortex M4, which has no data cache. However we also have a "countermeasure" version that stores the occupancy of v_i in list of a dozen 64-bit words (the sign doesn't matter, so a single bit is enough). This version scans the entire list with constant-time Boolean logic for every probe.

4 Performance on Cortex M4

We benchmarked a group of comparable NIST First Round KEM and PKE proposals on Cortex-M4. A NIST Category 3 variant ("192 quantum bit security") was used if available. We used an optimized C version in our tests, linked with an efficient assembler-language SHA3 implementation (excluded from code size). Our results are summarized in Table 2 and Figures 1 and 2. We are also including

Table 2. Communication parameters and cycle count breakdown of the optimized C implementation on Cortex-M4 for some NIST PQC candidate KEMs. First columns give the size of public key, secret key, and ciphertext in bytes. The following columns give the number of cycles required for key generation, encapsulation, and decapsulation.

Algorithm	Size in Bytes			Cycles (k=1000, M=10^6)		
	PK	**SK**	**CT**	**KeyGen**	**Encaps**	**Decaps**
`R5ND_1KEM`	538	16	632	647 k	920 k	355 k
`R5ND_3KEM`	780	24	904	1,026 k	1,437 k	504 k
`R5ND_5KEM`	1050	32	1207	1,376 k	1,919 k	637 k
`R5ND_1PKE`	562	594	672	545 k	847 k	1,067 k
`R5ND_3PKE`	810	858	1032	925 k	1,388 k	1,737 k
`R5ND_5PKE`	1140	1204	1376	1,245 k	1,823 k	2,229 k
Kyber-768 [2]	1088	2400	1152	1,333 k	1,765 k	1,935 k
NewHope1024CCA [1]	1824	3680	2208	1,505 k	2,326 k	2,493 k
Saber [8]	992	2304	1088	7,156 k	9,492 k	11,612 k
sntrup4591761 [4]	1218	1600	1047	166,215 k	11,274 k	31,733 k
NTRU-HRSS17 [16]	1138	1418	1278	187,525 k	5,429 k	15,405 k
SIKEp751 [17]	564	644	596	3,775 M	6,114 M	6,572 M

assembler optimized Cortex M4 numbers for Saber from [18] for completeness. Our implementation is available at `https://github.com/round5/r5nd_tiny`.

Test Setup. We wanted a fair comparison that eliminates bias caused by poor testing setup. For example the initial NIST tests of Hila5 indicated poor performance, but this was caused by extremely poor implementation of `randombytes()` in the NIST test suite. This function (i.e. the test suite itself) was consuming 80 % of cycles. Some other submitters were aware of this pitfall and created a faster layer of random number generation *inside* their implementation. Our test code consistently uses a fast implementation of SHAKE for random numbers.

Gnu C compiler `arm-none-eabi-gcc` was used with optimization flags set to `-Ofast -mthumb`. Our testing was performed on MXP MK20DX256 Microcontroller on a Teensy[6] board, which we ran at 24 MHz. Cycle counts at higher speeds are slightly less accurate due to interference by the memory controller.

For comparison, a highly optimized X25519 multiplication is 907 k cycles on Cortex M4 [12] – but has only 128-bit classical security. Four scalar multiplications are needed for Diffie-Hellman, so plain C implementations of `R5ND_1KEM`, `R5ND_3PKE`, and even higher-security `R5ND_3KEM` are faster.

Dominance of Hashing in KEM Speed Measurement. Our measurement results on other candidates are consistent with those produced by the

[6] Teensy 3.2 is an inexpensive (under $20) miniature (18 × 36 mm or 0.7 × 1.4") Cortex-M4 development board: `https://www.pjrc.com/store/teensy32.html`

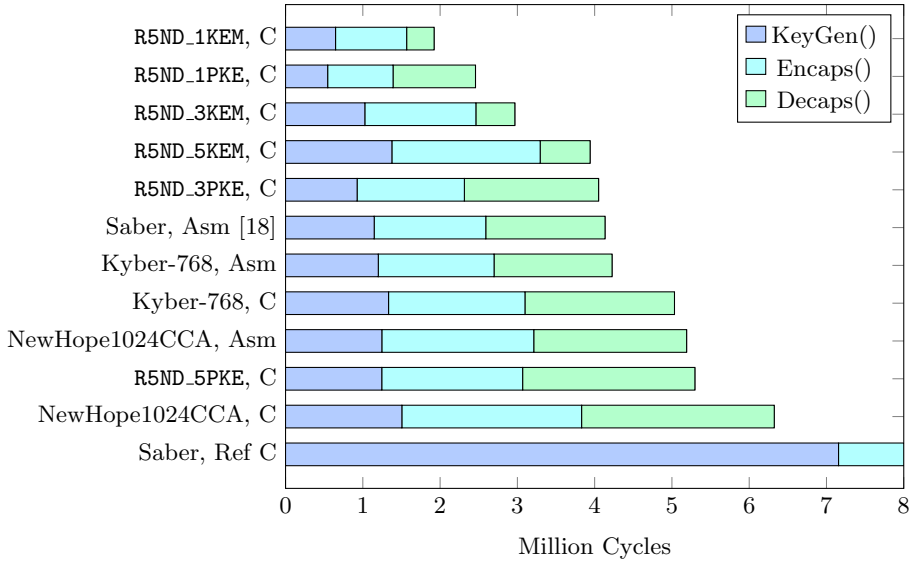

Fig. 2. Visualization of relative speed of key establishment, including assembler versions from [18] and PQM4. We have excluded algorithms that require more than 200 M cycles (several seconds).

PQCRYPTO group in "PQM4: Post-quantum crypto library for the ARM Cortex-M4" project[7]. However we didn't use their testing script system as it is targeted to a board using different flashing and communication mechanisms.

There were puzzling features in our initial results such as slightly *faster decapsulation than encapsulation* on some CCA algorithms – this is counter-intuitive since the Fujisaki-Okamoto Transform [13,15] requires that a full "simulated encryption" is included in every decapsulation operation.

The only reason for wide performance divergence was that PQM4 experiments used a hand-crafted assembler implementation of the SHA3 core. The optimized Sponge permutation takes 11,785 cycles, while its optimized C language equivalent takes 32,639 cycles. The "faster CCA decapsulation" anomaly was simply a manifestation of more hashing being required in encapsulation – a faster hash made CCA encapsulation faster again.

The performance difference caused by hash implementation is more pronounced in some candidates than others – 23% in our CCA candidate but as high 82% in case of Kyber-768. Therefore it is good to keep in mind that benchmarks of fast lattice KEMs are also benchmarks of the hash function implementation.

[7] PQM4 source code and results are available at `https://github.com/mupq/pqm4`

Table 3. Engineering and security comparison for key establishment use case with Cortex M4 at NIST Security Level 3. Xfer: Total data transferred (public key + ciphertext). Time: Time required for KeyGen() + Encaps() + Decaps() on Cortex-M4 at 24 MHz. Code: Size of implementation in bytes, excluding hash function and other common parts. Fail: Decryption failure bound. PQ Sec: Claimed quantum complexity. Classic: Claimed classical complexity.

Algorithm	Xfer	Time	Code	Fail	PQ Sec	Classic
R5ND_3KEM	1684	0.124s	4464	2^{-75}	2^{176}	2^{193}
R5ND_3PKE	1842	0.169s	5232	2^{-129}	2^{181}	2^{193}
Saber [8,18]	2080	0.172s	?	2^{-136}	2^{180}	2^{198}
Kyber-768 [2]	2240	0.210s	7016	2^{-142}	2^{161}	2^{178}
sntrup4591761 [4]	2265	8.718s	71024	0	?	2^{248}
NTRU-HRSS17 [16]	2416	7.814s	11956	0	2^{123}	2^{136}
NewHope1024-CCA [1]	4032	0.264s	12912	2^{-216}	2^{233}	?
SIKEp751 [17]	1160	685.9s	19112	0	2^{124}	2^{186}

5 Conclusions

In this work have examined the suitability of Round5 post-quantum key establishment and public key encryption algorithm for embedded and other limited-resource use cases. We focused on Cortex M4 implementation of R5ND_3KEM and R5ND_3PKE variants and compared them to some other compact NIST PQC proposals at the same security level.

Round5 combines the design features of two candidates in the NIST Post-Quantum Cryptography project, Round2 and Hila5. Round5 has new parameter selection, addressing various NIST PQC security levels and use cases. Optimization of parameters was performed primarily for bandwidth at given security level; the public key and ciphertext sizes of the new variant are smaller than those of other lattice candidates and second smallest only to SIKE (which is not practical on embedded targets due to its very high computational requirements).

Round5 relies on an error correcting code (based on that of Hila5) to further reduce failure probability, and thus allow parameters to be adjusted for even better bandwidth efficiency. There are many other new features and changes in relation to Round2, such as use of SHAKE-256 for deterministic pseudorandom sequence generation, and a new method for creating sparse ternary polynomials.

The avoidance of Number Theoretic Transform in multiplication helps to bring the implementation size down, but has raised questions about performance. We benchmarked the Round5 ring variants on a Cortex M4 microcontroller and found them to have equivalent, or significantly better performance characteristics than other comparable candidates. Table 3 offers an "engineering" comparison for a key establishment use case at NIST Category 3 security level, and shows why we see Round5 as a leading candidate, at least on embedded targets.

References

1. Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. NewHope: Algorithm specifcations and supporting documentation. First Round NIST PQC Project Submission Document, November 2017. URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.
2. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation. Fist Round NIST PQC Project Submission Document, November 2017. URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.
3. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, 2012. `doi:10.1007/978-3-642-29011-4_42`.
4. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. Ntru prime 20171130. Fist Round NIST PQC Project Submission Document, November 2017. URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.
5. Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. Submitted for publication, August 2018. URL: `https://eprint.iacr.org/2018/725`.
6. CNSS. Use of public standards for the secure sharing of information among national security systems. Committee on National Security Systems: CNSS Advisory Memorandum, Information Assurance 02-15, July 2015.
7. Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003. URL: `http://www.shoup.net/papers/cca2.pdf`, `doi:10.1137/S0097539702403773`.
8. Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Mod-LWR based KEM. Fist Round NIST PQC Project Submission Document, November 2017. URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.
9. Morris Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. NIST Special Publication 800-38A, December 2001. `doi:10.6028/NIST.SP.800-38A`.
10. FIPS. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, November 2001. URL: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
11. FIPS. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication 202, August 2015. `doi:10.6028/NIST.FIPS.202`.
12. Hayato Fujii and Diego F. Aranha. Curve25519 for the Cortex-M4 and beyond. In *LATINCRYPT 2017*, 2017. To appear. URL: `http://www.cs.haifa.ac.il/~orrd/LC17/paper39.pdf`.
13. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael Wiener, editor, *CRYPTO 1999*, volume 1666 of *LNCS*, pages 537–554. Springer, 1999. `doi:10.1007/3-540-48405-1_34`.

14. Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, Jose-Luis Torre-Arce, and Hayo Baan. Round2: KEM and PKE based on GLWE. First Round NIST PQC Project Submission Document, November 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.
15. Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017 (I)*, volume 10677 of *LNCS*, pages 341–371. Springer, 2017. URL: https://eprint.iacr.org/2017/604, doi:10.1007/978-3-319-70500-2_12.
16. Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. NTRU-HRSS-KEM: Algorithm specifications and supporting documentation. Fist Round NIST PQC Project Submission Document, November 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.
17. David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation. First Round NIST PQC Project Submission Document, November 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.
18. Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM. In *CHES 2018 – to appear*, 2018. URL: https://eprint.iacr.org/2018/682.
19. Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011. doi:10.1007/978-3-642-19074-2_21.
20. Dustin Moody. Post-quantum cryptography: NIST's plan for the future. Talk given at PQCrypto '16 Conference, 23-26 February 2016, Fukuoka, Japan, February 2016. URL: https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf.
21. NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Official Call for Proposals, National Institute for Standards and Technology, December 2016. URL: http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf.
22. NIST. Post-quantum cryptography – round 1 submissions. National Institute for Standards and Technology, December 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.
23. NSA/CSS. Information assurance directorate: Commercial national security algorithm suite and quantum computing FAQ, January 2016. URL: https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm.
24. Henri J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28:205–215, 1980. doi:10.1109/TASSP.1980.1163372.
25. John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information & Computation*, 3(4):317–344, July 2003. Updated version available on arXiv. URL: https://arxiv.org/abs/quant-ph/9508027.
26. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC '05*, pages 84–93. ACM, May 2005. doi:10.1145/1060590.1060603.

27. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, September 2009. doi:10.1145/1568318.1568324.
28. Markku-Juhani O. Saarinen. Ring-LWE ciphertext compression and error correction: Tools for lightweight post-quantum cryptography. In *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '17, pages 15–22. ACM, April 2017. URL: https://eprint.iacr.org/2016/1058, doi:10.1145/3055245.3055254.
29. Markku-Juhani O. Saarinen. HILA5: On reliability, reconciliation, and error correction for Ring-LWE encryption. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *Lecture Notes in Computer Science*, pages 192–212. Springer, 2018. doi:10.1007/978-3-319-72565-9_10.
30. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. FOCS '94*, pages 124–134. IEEE, 1994. Updated version available on arXiv. URL: https://arxiv.org/abs/quant-ph/9508027, doi:10.1109/SFCS.1994.365700.

Table 4. All Round5 parameter sets, with performance estimates, post-quantum and classical security levels, and failure rate. Ciphertext size for PKE variants does not include the overhead required for DEM (typically 16 bytes for an authentication tag). Quoted from [5]. **NOTE: PARAMETERS ARE BEING REVISED DUE TO RING CHANGE.**

		Round5.KEM			Round5.PKE		
	Parameters	**CPA NIST1**	**CPA NIST3**	**CPA NIST5**	**CCA NIST1**	**CCA NIST3**	**CCA NIST5**
$n = d$, **Ring variants.**	d, n, h	522, 522, 208	756, 756, 242	1018, 1018, 254	546, 546, 158	786, 786, 204	1108, 1108, 198
	q, p, t	$2^{14}, 2^8, 2^4$	$2^{15}, 2^8, 2^4$	$2^{15}, 2^8, 2^4$	$2^{16}, 2^8, 2^4$	$2^{16}, 2^8, 2^6$	$2^{16}, 2^8, 2^5$
	$B, \bar{n}, \bar{m}, f$	1, 1, 1, 3	1, 1, 1, 3	1, 1, 1, 3	1, 1, 1, 3	1, 1, 1, 3	1, 1, 1, 3
	μ	$128 + 91$	$192 + 103$	$256 + 121$	$128 + 91$	$192 + 103$	$256 + 121$
	Bandwidth	1170 B	1684 B	2257 B	1234 B	1842 B	2516 B
	Public key	538 B	780 B	1050 B	562 B	810 B	1140 B
	Ciphertext	632 B	904 B	1207 B	672 B	1032 B	1376 B
	PQ Security	2^{117}	2^{176}	2^{242}	2^{120}	2^{181}	2^{246}
	Classical	2^{128}	2^{193}	2^{257}	2^{128}	2^{193}	2^{256}
	Failure rate	2^{-76}	2^{-75}	2^{-64}	2^{-129}	2^{-128}	2^{-129}
	Version ($f^{(0)}_{d,d}$)	`R5ND_1KEM`	`R5ND_3KEM`	`R5ND_5KEM`	`R5ND_1PKE`	`R5ND_3PKE`	`R5ND_5PKE`
$n = 1$, **Non-ring variants.**	d, n, h	635, 1, 266	929, 1, 268	1186, 1, 712	694, 1, 152	932, 1, 540	1198, 1, 574
	q, p, t	$2^{15}, 2^{11}, 2^{10}$	$2^{14}, 2^{11}, 2^{10}$	$2^{14}, 2^{12}, 2^7$	$2^{13}, 2^{11}, 2^{10}$	$2^{14}, 2^{12}, 2^9$	$2^{14}, 2^{12}, 2^{10}$
	$B, \bar{n}, \bar{m}, f$	4, 6, 6, 0	4, 6, 8, 0	4, 8, 8, 0	4, 5, 7, 0	4, 6, 8, 0	4, 8, 8, 0
	μ	32	48	64	32	48	64
	Bandwidth	10535 B	17969 B	28553 B	11553 B	19703 B	28925 B
	Public key	5256 B	7690 B	14265 B	4789 B	8413 B	14409 B
	Ciphertext	5279 B	10279 B	14288 B	6764 B	11290 B	14516 B
	PQ Security	2^{119}	2^{182}	2^{233}	2^{122}	2^{176}	2^{233}
	Classical	2^{128}	2^{192}	2^{256}	2^{128}	2^{192}	2^{256}
	Failure rate	2^{-65}	2^{-65}	2^{-84}	2^{-128}	2^{-135}	2^{-129}
	Version ($f^{(0)}_{d,d}$)	`R5T0_1KEM`	`R5T0_3KEM`	`R5T0_5KEM`	`R5T0_1PKE`	`R5T0_3PKE`	`R5T0_5PKE`
	Version ($f^{(0)}_{d,d}$)	`R5T1_1KEM`	`R5T1_3KEM`	`R5T1_5KEM`	`R5T1_1PKE`	`R5T1_3PKE`	`R5T1_5PKE`
	Version ($f^{(0)}_{d,d}$)	`R5T2_1KEM`	`R5T2_3KEM`	`R5T2_5KEM`	`R5T2_1PKE`	`R5T2_3PKE`	`R5T2_5PKE`