Post-Quantum Key Exchange on ARMv8-A
A New Hope for NEON made Simple

Silvan Streit[1,2] and Fabrizio De Santis[1]

[1] Technische Universität München (TUM), Germany.
{silvan.streit,desantis}@tum.de
[2] Ludwig-Maximilians-Universität München (LMU), Germany.
silvan.streit@lmu.de

Abstract. NewHope and NewHope-Simple are two recently proposed post-quantum key exchange protocols based on the hardness of the Ring-LWE problem. Due to their high security margins and performance, there have been already discussions and proposals for integrating them into Internet standards, like TLS, and anonymity network protocols, like Tor. In this work, we present time-constant and vector-optimized implementations of NewHope and NewHope-Simple for ARMv8-A 64-bit processors which target high-speed applications. This architecture is implemented in a growing number of smart phone and tablet processors, and features powerful 128-bit SIMD operations provided by the NEON engine. In particular, we propose the use of three alternative modular reduction methods, which allow to better exploit NEON parallelism by avoiding larger data types during the Number Theoretic Transform (NTT) and remove the need to transform input coefficients into Montgomery domain during pointwise multiplications. The NEON vectorized NTT uses a 16-bit unsigned integer representation and executes in only $18,909$ clock cycles on an ARM Cortex-A53 core. Our implementation improves previous assembly-optimized results on ARM NEON platforms by a factor of 3.4 and outperforms the C reference implementation on the same platform by a factor of 8.3. The total time spent on the key exchange was reduced by more than a factor of 3.5 for both protocols.

Keywords: ARMv8-A, NEON, ARM Cortex-A53, Post-Quantum Key Exchange, Ring-LWE, NewHope, NewHope-Simple, NTT.

1 Introduction

The National Institute of Standards and Technology (NIST) has recently initiated the process of identifying and evaluating post-quantum public-key cryptographic algorithms for the upcoming future, when large quantum computers might be constructed [15]. Yet, the immediate deployment of post-quantum algorithms, along with elliptic-curve or RSA based algorithms, is necessary to avoid the risk of current network traffic being recorded and broken in the future. A first passively secure key exchange protocol based on the Ring Learning with Errors problem (Ring-LWE) employing an error reconciliation mechanism was proposed by Ding, Xie, and Lin in [16, 17]. A tweaked version of it was later on put forward by Chris Peikert in [26] as a "*drop-in component*" for Internet standard protocols. A first instantiation of this protocol with concrete parameters was then proposed by Bos, Costello, Naehrig, and Stebila in [13] and implemented into the Transport Layer Security (TLS) protocol of OpenSSL. A further improvement was suggested by Alkim, Ducas, Pöppelmann, and Schwabe in [3], also referred to as NewHope. Some of the main improvements of [3] over the previous proposal [13] are the simplified noise distribution, which is a centered binomial noise distribution, and an improved selection of the ring parameters geared towards higher performance and security. Real world applications of NewHope were tested by Google [20] and an integration into the anonymity network Tor was proposed in [14]. Finally, a very recent update was made to the protocol and called NewHope-Simple, which replaces the reconciliation mechanism by a simple key encapsulation method [2].

State-of-the-Art Implementations The authors of NewHope provide a portable reference `C` implementation, as well as an assembly-optimized version targeting high-end Intel Haswell processors using AVX2 vector operations[3]. Gueron and Schlieker [21] further parallelized the sampling step using AVX2 and proposed techniques to reduce the rejection rate of pseudorandom numbers, which was also independently proposed by Yawning Angel [5]. The implementation of the Number Theoretic Transform (NTT) using AVX2 instructions was later on optimized by Longa and Naehrig in [23], using an alternative modular reduction technique and a signed representation of integers. NewHope was ported to ARM Cortex-M embedded processors by Alkim, Jakubeit, and Schwabe in [4]. They reduced the memory accesses in the NTT and optimized subroutines in assembly to gain improved performance from the available instruction set.

Our Contribution All implementations of NewHope so far target either high-end desktop (e.g. AVX2) or low-end embedded processors (e.g. ARM Cortex-M). In this work, we close this gap by presenting constant-time[4] assembly-optimized implementations of NewHope and NewHope-Simple protocols for ARMv8-A 64-bit processors. This architecture includes the NEON SIMD vector extension by default, which has been already successfully used many times in the past to speed up a variety of cryptographic algorithms [11], including other Ring-LWE based algorithms [7]. In particular, we fully vectorized all ring operations, while optimizing pipelining. Further, we use alternative reductions which are more efficient and suited for NEON ARMv8-A SIMD than previously used methods. For the computation of the NTT, an improved fast reduction mechanism is proposed, which is similar to Barrett reduction, but does not need to extend intermediate values to larger data types. A full Barrett reduction routine is used during the pointwise multiplication of two polynomials, removing the need to transform the coefficients into Montgomery domain. Further, a reduction "by minimum" is proposed for addition and subtraction.

[3] `https://cryptojedi.org/crypto/#newhope`

[4] The generation of public parameters is not constant-time [3].

2 NewHope and NewHope-Simple

NewHope is an ephemeral key exchange protocol proposed by Alkim, Ducas, Pöppelmann, and Schwabe [3] in 2015 and aimed at 128-bit post-quantum security. An overview of NewHope is provided in Figure 1. The protocol performs computations in the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 1024$ and $q = 12289$. To enable efficient computations in $\mathcal{R}_q$, NewHope uses the negacyclic Number Theoretic Transform (NTT) to transform polynomials into Fourier space. Let $\mathbf{a} = \sum_{i=0}^{1023} a_i X^i \in \mathcal{R}_q$, then

$$\hat{\mathbf{a}} = \mathsf{NTT}(\mathbf{a}) = \sum_{i,j=0}^{1023} \gamma^j a_j \omega^{ij} X^i, \tag{1}$$

with $\omega = 49$ being the 1024th primitive root of unity and $\gamma = \sqrt{\omega} = 7$ being the twiddle factor of the negacyclic NTT. The inverse transform NTT^{-1} is defined as:

$$\mathbf{a} = \mathsf{NTT}^{-1}(\hat{\mathbf{a}}) = \sum_{i,j=0}^{1023} n^{-1} \gamma^{-i} \hat{a}_j \omega^{-ij} X^i. \tag{2}$$

These transforms allow to compute the polynomial multiplication $\mathbf{a} * \mathbf{b}$ simply as $\mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$, thus replacing expensive polynomial multiplications (denoted by $*$) with $\mathsf{NTT}/\mathsf{NTT}^{-1}$ and pointwise multiplications (denoted by $\circ$). In NewHope, two entities $\mathcal{A}$ and $\mathcal{B}$ agree on a shared secret in three steps:

1. $\mathcal{A}$ initiates the protocol via an $\mathsf{offer}()$ to $\mathcal{B}$ by computing a public polynomial $\hat{\mathbf{b}}$ from a 256-bit seed σ and a secret polynomial $\hat{\mathbf{s}}$. First, $\mathsf{Parse}()$ generates a random polynomial $\hat{\mathbf{a}}$ using a variable amount of output of $\mathsf{SHAKE\text{-}128}$ [18] stretched from σ. Then, the coefficients of the secret polynomial $\mathbf{s}$ and the error polynomial $\mathbf{e}$ are sampled at random from a centered binomial distribution $\Psi_{16} = \sum_{i=0}^{15} b_i - b'_i$ using uniform random bits $b_i, b'_i \in \{0,1\}$. Finally, the public polynomial $\hat{\mathbf{b}}$ is obtained by multiplying the NTT-transformed secret polynomial $\hat{\mathbf{s}}$ by $\hat{\mathbf{a}}$ and adding the NTT of the error polynomial $\mathbf{e}$ to the result. The public seed σ and the polynomial $\hat{\mathbf{b}}$ are encoded into a byte-string m_A via $\mathsf{encodeA}()$, and sent over to $\mathcal{B}$.
2. $\mathcal{B}$ performs an $\mathsf{accept}()$ to derive a second public polynomial $\hat{\mathbf{u}}$, a helper polynomial $\mathbf{r} \in \mathcal{R}_4$, and a shared secret $\mu \in \{0,1\}^{256}$. First, the polynomials $\mathbf{t}$,$\mathbf{e}'$, and $\mathbf{e}''$ are sampled from Ψ_{16}. By decoding m_A back to $(\hat{\mathbf{b}}, \sigma)$ via $\mathsf{decodeA}()$, $\hat{\mathbf{a}}$ is recreated via $\mathsf{Parse}()$. The public polynomial $\hat{\mathbf{u}}$ is derived in a similar way as for $\hat{\mathbf{b}}$, by multiplying $\hat{\mathbf{t}}$ by $\hat{\mathbf{a}}$ and adding $\mathsf{NTT}(\mathbf{e}')$ to the result. Then, the public polynomial $\hat{\mathbf{b}}$ is used to generate the shared polynomial $\mathbf{v}$ by multiplying it with the secret polynomial $\hat{\mathbf{t}}$, applying the NTT^{-1} to the result, and finally adding the error polynomial $\mathbf{e}''$. Finally, the shared secret μ is computed by reconciliation as follows: The helper polynomial $\mathbf{r}$ is generated via $\mathsf{HelpRec}()$ from the shared polynomial $\mathbf{v}$. With this helper data and $\mathbf{v}$ a common secret value ν is reconstructed via $\mathsf{Rec}()$, and finally whitened using $\mathsf{SHA3\text{-}256}$. The public polynomial $\hat{\mathbf{u}}$ along with the helper data $\mathbf{r}$ is byte-wise encoded via $\mathsf{encodeB}()$, and sent back to $\mathcal{A}$.

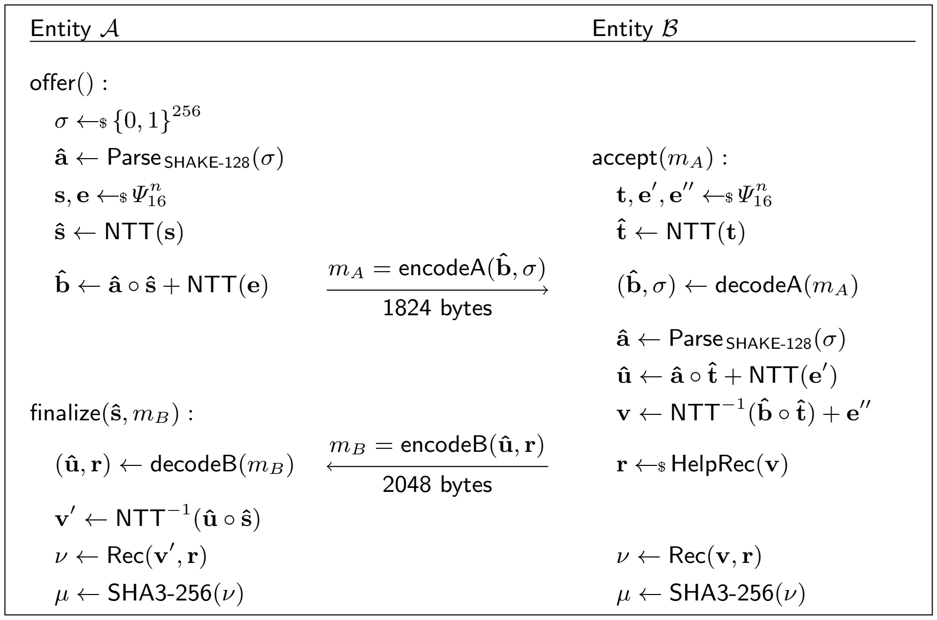
Entity $\mathcal{A}$
Entity $\mathcal{B}$
offer() :
$\sigma \leftarrow_{\$} \{0,1\}^{256}$
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}_{\mathsf{SHAKE\text{-}128}}(\sigma)$
$\mathbf{s}, \mathbf{e} \leftarrow_{\$} \Psi_{16}^{n}$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$
$m_A = \mathsf{encodeA}(\hat{\mathbf{b}}, \sigma) \xrightarrow[\text{1824 bytes}]{}$
accept(m_A) :
$\mathbf{t}, \mathbf{e}', \mathbf{e}'' \leftarrow_{\$} \Psi_{16}^{n}$
$\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{t})$
$(\hat{\mathbf{b}}, \sigma) \leftarrow \mathsf{decodeA}(m_A)$
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}_{\mathsf{SHAKE\text{-}128}}(\sigma)$
$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
$\mathbf{v} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$
finalize($\hat{\mathbf{s}}, m_B$) :
$(\hat{\mathbf{u}}, \mathbf{r}) \leftarrow \mathsf{decodeB}(m_B)$
$m_B = \mathsf{encodeB}(\hat{\mathbf{u}}, \mathbf{r}) \xleftarrow[\text{2048 bytes}]{}$
$\mathbf{r} \leftarrow_{\$} \mathsf{HelpRec}(\mathbf{v})$
$\mathbf{v}' \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$
$\nu \leftarrow \mathsf{Rec}(\mathbf{v}', \mathbf{r})$
$\nu \leftarrow \mathsf{Rec}(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$

Fig. 1. The NewHope protocol [3].

3. $\mathcal{A}$ derives the shared secret μ from $(\hat{\mathbf{s}}, m_B)$ via finalize() also by reconciliation. It multiplies $\hat{\mathbf{u}}$ by $\hat{\mathbf{s}}$ to obtain almost an identical copy of $\mathbf{v}$ which is denoted by $\mathbf{v}'$. Finally, it uses the helper polynomial $\mathbf{r}$ to reconciliate $\mathbf{v}'$ back to ν via Rec(), and hashes the result to obtain μ.

Very recently, a simplified version of NewHope was proposed by Alkim, Ducas, Pöppelmann, and Schwabe in [2] under the name of NewHope-Simple. Compared to the original proposal, the reconciliation mechanism is replaced by a simple Ring-LWE encryption, as shown in Figure 2. The cost of this simplification is an increase in size of the reply message to 2176 bytes (6.25% more). Further, $\mathcal{B}$ generates a 256-bit random key ν, in contrast to the 256 random bits used in HelpRec() in NewHope. The common secret ν is hashed using SHA3-256 and each bit is encoded into the upper bits of four coefficients of $\mathbf{k}$ in NHSEncode(). The ciphertext polynomial $\mathbf{c}$ is computed by a pointwise multiplication of $\hat{\mathbf{b}}$ and $\hat{\mathbf{t}}$, transformed back to normal domain with NTT^{-1} followed by an addition of $\mathbf{e}''$ and $\mathbf{k}$. It is then transferred compressed to $\mathcal{A}$ using NHSCompress(), removing the unnecessary lower bits by modulus switching and restored on the initiators side as $\mathbf{c}'$ with NHSDecompress(). By subtracting the NTT^{-1} of the product $\hat{\mathbf{u}} \circ \hat{\mathbf{s}}$ from $\mathbf{c}'$, $\mathcal{A}$ is able to restore an approximate copy of $\mathbf{k}$. By combining each four coefficients the hashed key ν' is extracted in NHSDecode(). Similar to the original NewHope protocol the common secret ν is finally hashed to the shared secret μ using SHA3-256().

Entity $\mathcal{A}$		Entity $\mathcal{B}$
offer() :		
$\sigma \leftarrow_{\$} \{0,1\}^{256}$		
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}_{\mathsf{SHAKE\text{-}128}}(\sigma)$		accept(m_A) :
$\mathbf{s}, \mathbf{e} \leftarrow_{\$} \Psi_{16}^{n}$		$\mathbf{t}, \mathbf{e}', \mathbf{e}'' \leftarrow_{\$} \Psi_{16}^{n}$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$		$\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{t})$
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$	$\xrightarrow[\text{1824 bytes}]{m_A = \mathsf{encodeA}(\hat{\mathbf{b}}, \sigma)}$	$(\hat{\mathbf{b}}, \sigma) \leftarrow \mathsf{decodeA}(m_A)$
		$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}_{\mathsf{SHAKE\text{-}128}}(\sigma)$
		$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
		$\nu \leftarrow_{\$} \{0,1\}^{256}$
		$\nu' \leftarrow \mathsf{SHA3\text{-}256}(\nu)$
		$\mathbf{k} \leftarrow \mathsf{NHSEncode}(\nu')$
finalize$(\hat{\mathbf{s}}, m_B)$:		$\mathbf{c} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mathbf{k}$
$(\hat{\mathbf{u}}, \bar{\mathbf{c}}) \leftarrow \mathsf{decodeB}(m_B)$	$\xleftarrow[\text{2176 bytes}]{m_B = \mathsf{encodeB}(\hat{\mathbf{u}}, \bar{\mathbf{c}})}$	$\bar{\mathbf{c}} \leftarrow \mathsf{NHSCompress}(\mathbf{c})$
$\mathbf{c}' \leftarrow \mathsf{NHSDecompress}(\bar{\mathbf{c}})$		
$\mathbf{k}' \leftarrow \mathbf{c}' - \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		
$\nu' \leftarrow \mathsf{NHSDecode}(\mathbf{k}')$		
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu')$		$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu')$

Fig. 2. The NEWHOPE-SIMPLE protocol [2].

3 ARMv8-A Architecture

ARMv8-A is a 64-bit RISC architecture with 31 general purpose registers (`r0` to `r30`) [1]. These can either contain 64-bit values (denoted by `Xn`) or 32-bit values (denoted by `Wn`). The ARMv8-A architecture further features 32× SIMD and floating point 128-bit registers (`v0` to `v31`). For non-vector operations these registers can be referenced by the number of bits used in the computation (`Bn` for 8-bit, `Hn` for 16-bit, `Sn` for 32-bit, `Dn` for 64-bit, `Qn` for 128-bit). When accessed as vectors, the registers are referenced as `vn.xy`, with `n` being the register number, `x` being the number of elements, and `y` being the element size encoded in the letters according to the non-vector access. Single elements of a vector can be accessed by square brackets, e.g. `v0.h[2]` for the third half-word of `v0`.

The NEON extension offers very powerful vector SIMD operations which process the elements of input vectors in parallel. They include the regular arithmetic operations `ADD`, `SUB` and `MUL` as well as bitwise operations like `AND` and `ORR`. For multiplications followed by an addition it offers multiply and accumulate (`MLA`) as well as multiply and subtract (`MLS`) instructions. Further, long multiplications are available (`UMULL/2`), resulting in the output elements being twice as large as the inputs. Also, wide operations are supported, with one input vector consisting of half-width elements compared to the output, e.g. signed wide addition (`SADDW/2`). Values can be narrowed to half-width data-types using extract narrow (`XTN/2`). Along with regular shifts, like unsigned right shift (`USHR`), it also offers advanced shifts, e.g. shift and accumulate (`USRA`) as well as narrowing shifts (`SHRN/2`). Finally, `UMIN` instruction selects the minimum of each element out of two input vectors, while `CNT` counts the number of '1's in each byte. Loading of vectors can be done using `LD1`, which loads consecutive elements into one vector, as well as `LD2`, `LD3` and `LD4` which load consecutive elements de-interleaved into 2, 3 and 4 vectors respectively. The elements of a vector can be also easily reordered, e.g. by `TRN1/2`, which merge either even or odd elements of two vectors, respectively.

4 Implementation Details

In this section, implementation details about $\mathsf{NTT}/\mathsf{NTT}^{-1}$, pointwise multiplication ($\circ$), polynomial addition ($+$), polynomial subtraction ($-$), as well as noise sampling from Ψ_{16}^{n} are discussed. In particular, the proposed reduction methods are elaborated.

4.1 NTT and NTT^{-1}

The NTT as well as its inverse NTT^{-1} (see Equations (1) and (2)) can be implemented efficiently using Gentleman-Sande (GS) butterfly operations [19] in a Fast Fourier Transform (FFT) structure. For $n = 1024$, the NTT can be computed in 10 levels, each consisting of 512 GS-butterfly operations which combine two polynomial coefficients by the means of an addition, and a subtraction followed by a multiplication with a power of ω. This excludes the multiplication by the powers of γ for the NTT as well as the powers of γ^{-1} and n^{-1} for the NTT^{-1}, which are performed before the NTT and after the NTT^{-1}. GS-butterfly based $\mathsf{NTT}/\mathsf{NTT}^{-1}$ generally need the input polynomial to be in bit-reversed ordering. However, in NewHope the bit-reversal of the NTT is not performed, as it is only applied to randomly generated polynomials.

Parallelism vs Laziness Trade-Off Evaluation The coefficients of $\mathcal{R}_{12289}$ are integers $< 2^{14}$. The ARMv8-A NEON supports SIMD instructions on $8\times$ 16-bit, $4\times$ 32-bit, or $2\times$ 64-bit unsigned integers. When performing operations on two coefficients, the size of the result can increase till a certain size before being reduced. This allows for different trade-offs depending on the size of elements and the laziness of reductions during GS operations. Using smaller values allows for higher parallelism, but requires more reductions. On the other hand, using larger values allows for less parallelism, but increased laziness in the reductions. Our investigations suggest that 16-bit unsigned integers representation is the best choice in terms of speed for ARMv8-A NEON platforms, as the parallelism outweighs the advantage of lazy reductions.

Listing 1 GS-Butterfly Operation for 14-bit Inputs.

```
    Input:   va.8h = [$a_0$,...,$a_7$], vb.8h = [$b_0$,...,$b_7$],
             v$\omega$.8h = [$\omega_0$,...,$\omega_7$], vq.8h = [3q,...,3q].
    Output:  vc.8h = [$c_0$,...,$c_7$], vd.4s = [$d_0$,...,$d_3$], ve.4s = [$e_0$,...,$e_3$].
 1: ADD      ve.8h, va.8h, vq.8h              ▷ $e = a + 3q$
 2: SUB      ve.8h, ve.8h, vb.8h              ▷ $e = e - b$
 3: ADD      vc.8h, va.8h, vb.8h              ▷ $c = a + b$
 4: UMULL    vd.4s, ve.4h, v$\omega$.4h       ▷ $d = e \cdot \omega$
 5: UMULL2   ve.4s, ve.8h, v$\omega$.8h       ▷ $e = e \cdot \omega$
```

Merging of Levels Using 16-bit integers and having 32 registers available, then 256 values can be loaded. This allows merging of up to 7 levels of the NTT, which requires loading 128 coefficients and 64 powers of ω, in order to reduce the total number of load and store instructions. As the number of different powers of ω needed in the levels of the NTT decreases for higher levels, an asymmetrical split in two blocks is favorable, with the first consisting of 4 levels and the second of 6 levels. Using this approach all the 32 powers of ω for the second block can be loaded during the computation and thus do not need to be reloaded in each iteration. For the first block, each iteration needs a different selection of the 512 available powers of ω, thus they are loaded for each level separately during each iteration. Constants needed for the reduction routines are stored in three vector registers, removing the need to transfer them from standard registers for each reduction. Each block is iterated 16 times performing GS-butterfly operations on 64 coefficients, which are loaded at the beginning and stored at the end of the block, thus reducing the number of load and store instructions to $2n$, each.

Gentleman-Sande Butterfly GS-butterfly operations are applied to pairs of coefficient with a distance equal to the powers of 2, i.e. from 2^0 to 2^9 with each level. Hence, in the first level, neighboring values need to be combined, in order to enable vector operations between them. Thus, they are loaded de-interleaved with LD2. For the second level, transposing with TRN1/2 rearranges the elements to align each coefficient with its second neighboring coefficient in two consecutive vectors. Applying TRN1/2 to blocks of elements, aligns the coefficients in the consecutive levels. In the first level and all other odd levels, the result after addition is $< 2^{15}$ as the input coefficients $< 2^{14}$. In the remaining even levels, the inputs are $< 2^{15}$ and the output is $< 2^{16}$ and is reduced to $< 2^{14}$ using a fast Barrett reduction. In case of subtraction, the inputs are always $< 2^{15}$ for all levels, hence the addition of $2^{15} < 3q < 2^{16}$ makes the results always positive $< 2^{17}$. In contrast to [3], this work takes advantage of the fact that the inputs in odd levels are further $< 2^{14}$, and thus the result of subtraction is $< 2^{16}$ which allows further NEON parallelism in all odd levels, as shown in Listing 1. In all levels, the powers of ω are precomputed in Montgomery domain and multiplied by the result of subtractions. This allows for efficient Montgomery reductions [25].

Listing 2 Vectorized Montgomery Reduction.

```
   Input:    va.4s = [a_0,...,a_3], vb.4s = [b_0,...,b_3],
             vq.4s = [q,q',...], vr.4s = [r-1,...,r-1].
   Output:   vc.8h = [c_0,...,c_7].
1: MUL     vc.4s,  va.4s,  vq.s[1]          ▷ $c \leftarrow a \cdot q'$
2: AND     vc.16b, vc.16b, vr.16b           ▷ $c \leftarrow c \mod r$
3: MLA     va.4s,  vc.4s,  vq.s[0]          ▷ $a \leftarrow a + c \cdot q$
4: MUL     vc.4s,  vb.4s,  vq.s[1]          ▷ $c \leftarrow b \cdot q'$
5: AND     vc.16b, vc.16b, vr.16b           ▷ $c \leftarrow c \mod r$
6: MLA     vb.4s,  vc.4s,  vq.s[0]          ▷ $b \leftarrow b + c \cdot q$
7: SHRN    vc.4h,  va.4s,  16               ▷ $c = a \div 2^{16}$
8: SHRN2   vc.8h,  vb.4s,  16               ▷ $c = b \div 2^{16}$
9: USHR    vc.8h,  vc.8h,  2                ▷ $c = c \div 2^{2}$
```

Montgomery Reduction By using fully reduced powers of ω, the inputs of Montgomery reduction are limited by $\leq (2^{15} - 1 + 3 \cdot q) \cdot (q - 1) = 855,662,592 \approx 2^{29.67}$. Equivalent to [3], only 32-bit intermediate values are used and the parameter selected as $r = 2^{18}$. This restricts the input range to $< 2^{32} - q \cdot (r - 1) = 1,073,491,969 \approx 2^{29.99}$, limited by the final addition during the reduction. In Listing 2, a vectorized Montgomery reduction for ARMv8-A is shown. It takes two vectors with 4×32-bit elements each as an input and merges them in one vector with 8×16-bit elements, with values reduced to 14-bit in 9 instructions.

Longa-Naehrig Reduction In [23], an alternative method for modular reductions by expressing the modulus as $q = 12289 = 3 \cdot 2^{12} + 1$ is shown. For larger data types this approach offers performance enhancements. However, our choice of using 16-bit intermediates causes it to be less efficient than Montgomery reductions, as it needs two iterations to reduce 32-bit inputs to a 16-bit output.

Fast 16-bit Barrett Reduction The reduction after addition in every second level was implemented using fast 16-bit Barrett reduction. In [3] an extension to 32-bit is required by multiplying the input by a factor $A = 5$ before division. We propose the use of an improved version for ARMv8-A NEON, without the need to extend the intermediate value to 32-bit, as shown in Listing 3. The initial division by $r_1 = 2^3$ ensures 16-bit intermediate values after the multiplication of $A = 5$, while limiting the output to $\leq 16379 < 2^{14}$. Furthermore, the multiplication

Listing 3 Fast 16-bit Barrett Reduction.

```
   Input:    va.8h = [a_0,...,a_7], vq.8h = [q,...].
   Output:   va.8h = [a_0,...,a_7].
1: USHR   vt.8h, va.8h, 3                   ▷ $t \leftarrow a \div 2^{3}$
2: USRA   vt.8h, va.8h, 1                   ▷ $t \leftarrow t \cdot 5$
3: USHR   vt.8h, vt.8h, 13                  ▷ $t \leftarrow t \div 2^{13}$
4: MLS    va.8h, vt.8h, vq.h[0]             ▷ $a \leftarrow a - t \cdot q$
```

was replaced by a shift and accumulate operation to avoid the need of storing a constant in register. This approach is similar to the original version used by Paul Barrett in 1986 [8], as well as to the SAMS2 approach taken in [27] [7], but specialized for NewHope.

Bit-Reversal For NTT^{-1} an initial reordering step is needed. Due to its non-consecutive access vectorization is not applicable on ARMv8-A. Therefore, it was optimized by unrolling the swapping of elements in assembly.

Multiplication by $\boldsymbol{\gamma}$ The multiplication of the coefficients with the powers of γ and γ^{-1} for the NTT and NTT^{-1} was vectorized using NEON instructions. It uses precomputed powers of γ in the Montgomery domain, further including n^{-1} for the NTT^{-1}. A vectorized Montgomery reduction is applied afterwards.

4.2 Pointwise Multiplication $\circ$

This operation multiplies the coefficients of two vectors with each other in a pointwise way, i.e. each element only with the corresponding element of the other vector. It was vectorized, and further optimized for pipelining, by performing the pointwise multiplication of 2×64 coefficients in an interleaved order in 16 loop iterations. With input values up to 16-bit, this results in 32-bit output values.

In [3], two Montgomery reductions are used to first transform one input polynomial to Montgomery domain and later reduce the final result. Instead, we propose the use of a full 32-bit Barrett reduction, which removes the need of domain transformation and thus only needs a single iteration.

Full 32-bit Barrett Reduction Selecting $A = 2,863,078,533$ and $r = 2^{45}$ for the Barrett reduction $a' = a - \lfloor (a \cdot A) \div r \rfloor \cdot q$ yields a full reduction for all 32-bit unsigned integers. However, this requires extending the intermediate data type to 64-bit values and therefore needs 14 instructions on ARMv8-A NEON to reduce 8×32-bit values.

A Note on 28-bit Barrett Reduction As an alternative for platforms without efficient long multiplications for 32-bit values, we propose a fast 28-bit Barrett Reduction similar to our 16-bit version. Selecting $A = 43687$, $r_1 = 2^{12}$ and $r_1 = 2^{17}$ in $a' = a - \lfloor (\lfloor a \div r_1 \rfloor \cdot A) \div r_2 \rfloor \cdot q$ reduces values $\leq 17006 < 2q < 2^{15}$. Thus, an extra reduction step is necessary to limit the output to 14-bit, as well as care has to be taken to not exceed the 28-bit input range. On ARMv8-A NEON it needs 10 instructions to reduce 8×28-bit values to 15-bit.

4.3 Polynomial Addition $+$ and Subtraction $-$

Polynomial addition and subtraction calculate the sum and difference of two vectors element-wise, respectively. They were vectorized and interleaved for pipelining iterating over 2×64 coefficients in a loop. For the subtraction an extra factor

Listing 4 Vectorized Reduction by Minimum.

```
   Input:    va.8h = [a_0,...,a_7], vq.8h = [q,...,q].
   Output:   vb.8h = [b_0,...,b_7].
1: SUB    vb.8h, va.8h, vq.8h          ▷ $b = a - q$, underflow for $a < q$
2: UMIN   vb.8h, va.8h, vb.8h          ▷ $b = \min(a, b)$, unsigned
```

of $2q$ was added to avoid negative output values, together with the minuend always being fully reduced, the output is limited to $< 3q$. For the addition the input values are limited to 14-bit and thus the output to 15-bit $< 3q$. Hence, both can be reduced by two reductions of a single factor of q each. This results in a full reduction, removing the need to further reduce the polynomials in `encodeA()` and `encodeB()`.

Reduction by Minimum This reduction is based on the instruction `UMIN`, which selects the minimum element out of two vectors. It needs only two instructions in order to reduce 8× 16-bit vector elements, as shown in Listing 4.

4.4 Noise Sampling from Ψ_{16}^n

Similar to the reference `C` implementation, the seed is taken from `/dev/urandom` and extended using `ChaCha20` [9]. An ARMv8-A NEON optimized version of `ChaCha20` was used from the Linux Kernel development branch [12]. An alternative would have been to use AES-CTR by employing the AES instructions in ARMv8-A. However, this would have required the optional Cryptographic Extension to be integrated into the processor, which was not available on our testing platform.

The uniform noise is converted to the centered polynomial distribution Ψ_{16}^n by bit-wise accumulation followed by addition and subtraction using NEON vector operations. Together with a final addition of q in order to avoid negative output, 16 noise coefficients are generated in 9 instructions, as shown in Listing 5.

Listing 5 Vectorized Noise Sampling from Ψ_{16}^n.

```
   Input:    va.16b, vb.16b, vc.16b, vd.16b ←$ {0,1}^128, vq.8h = [q,...,q]
   Output:   ve.8h, vf.8h ←$ $\Psi_{16}^8$
1: CNT      va.16b, va.16b                       ▷ Sum up each 8-bit
2: CNT      vb.16b, vb.16b
3: CNT      vc.16b, vc.16b
4: CNT      vd.16b, vd.16b
5: ADD      va.16b, va.16b, vb.16b               ▷ Add two bytes
6: ADD      vb.16b, vc.16b, vd.16b
7: SUB      va.16b, va.16b, vb.16b               ▷ Subtract two bytes
8: SADDW2   vf.8h,  vq.8h,  va.16b               ▷ Add $q$ & extend to 16-bit
9: SADDW    ve.8h,  vq.8h,  va.8b
```

Table 1. Cycle count of NewHope and NewHope-Simple on ARM Cortex-A53.

Operation	`C` **Reference** [Cycles]	**NEON Assembly-Optimized** [Cycles]
NTT	156,564	18,909
NTT^{-1}	165,325	21,054
Pointwise Multiplication $\circ$	31,814	2,526
Polynomial Addition $+$	14,408	1,505
Polynomial Subtraction $-$	25,672	1,590
Noise Sampling Ψ_{16}^{n}	50,556	22,338
NewHope offer()	555,328	165,956
NewHope accept()	846,013	243,199
NewHope finalize()	220,141	47,027
NewHope-Simple offer()	555,301	166,028
NewHope-Simple accept()	853,453	238,478
NewHope-Simple finalize()	237,893	41,048

5 Performance Results

The performance was measured on an Odroid-C2 single-board computer [22] running Arch Linux ARM [6]. This board features an Amlogic S905 ARM Cortex-A53 1.5GHz quad-core CPU with 32KB L1 cache per core and 512KB L2 cache as well as 2GByte DDR3 SDRAM. Each ARM Cortex-A53 core implements the ARMv8-A RISC architecture with an 8-stage pipeline. The binaries were compiled with `gcc` 6.2.1 using the flags `-O3 -fomit-frame-pointer -march=native`. Performance was measured using the Linux kernel performance monitoring system call [24]. This enables user processes to accurately measure the performance on a multi-core platform using hardware counters. The measurements include only the cycles spent on the specific process and thus excludes all kernel interrupts and time spent by other processes on the same core. The reported cycle counts represent the median of 2^{16} consecutive measurements excluding the overheads of system calls. Table 1 provides a comparison between our NEON assembly-optimized version and the `C` reference implementation by [3] for NewHope and our modifications for NewHope-Simple. Our assembly-optimized implementation of the NTT (including a multiplication with γ) is faster by a factor of 8.3 compared to the `C` reference implementation. This is comparable to the speedup by a factor of 6.6 achieved with AVX2 by Alkim, Ducas, Pöppelmann, and Schwabe in [3]. Note that the AVX2 processor extension features 256-bit SIMD registers, while our platform features only 128-bit SIMD registers. Pointwise multiplication, as well as the polynomial addition and subtraction, were improved by factors between 9.6 and 16.1, when compared to the non-vectorized versions. A direct comparison with previous optimizations of Ring-LWE on the same platform is not applicable, as there are no optimized implementations known to the authors at the moment of writing. However, a

Table 2. Cycle Count of the NTT on ARM NEON Platforms.

Architecture	**Processor**	**Ring**	**Cycles**	
ARMv7-A [7]	Cortex-A9	$\mathbf{Z}_{7681}[x]/(x^{256}+1)$	$25,574$	$127,870^{\dagger}$
ARMv7-A [27]	Cortex-A9	$\mathbf{Z}_{12289}[x]/(x^{512}+1)$	$62,160$	$138,133^{\dagger}$
ARMv8-A [TW]	**Cortex-A53**	$\mathbf{Z_{12289}[x]/(x^{1024}+1)}$	$\mathbf{18,909}$	-

† Scaled to $n = 1024$ for comparison.

Table 3. Cycle Count of Ephemeral Key-Exchange on ARM NEON Platforms.

Architecture	**Processor**	**Protocol**	**$\mathcal{A}$-Cycles**	**$\mathcal{B}$-Cycles**
ARMv7-A [10]	Cortex-A9	X25519, NEON	$1,144,299^{\dagger}$	$1,144,299^{\dagger}$
ARMv8-A [10]	Cortex-A53	X25519, C ref.	$952,022^{\dagger}$	$952,022^{\dagger}$
ARMv8-A [TW]	**Cortex-A53**	**NewHope**	$\mathbf{212,983}$	$\mathbf{243,199}$
ARMv8-A [TW]	**Cortex-A53**	**NewHope-Simple**	$\mathbf{207,076}$	$\mathbf{238,478}$

† Computed as the sum of key pair generation and shared secret computation.

comparison with previously optimized implementations of the NTT on *other* NEON platforms is provided in Table 2. The numbers from [7, 27] are scaled by a factor of 5 and 20/9 considering the used dimensions $n = 256$ and $n = 512$, respectively. However, the experimental platform ARM Cortex-A9 employed in these implementations is less powerful and thus a comparison has to be done with care, e.g. the NEON extension on ARM Cortex-A9 only processes 64-bit of the 128-bit vector registers at a time. Therefore, if we consider a roughly estimated scaling factor of 2, our implementation outperforms these implementations by a factor of 3.4. Finally, for offer(), accept() and finalize() the performance is improved by a factor of 3.3 to 4.7 for both flavors. The NewHope-Simple finalize() shows more improvement with a factor 5.8, due to optimization of the polynomial subtraction used. The total key-exchange takes $212,983$ cycles for $\mathcal{A}$ and $243,199$ cycles for $\mathcal{B}$ for NewHope as well as $207,076$ and $238,478$ for NewHope-Simple, respectively. This also includes a final hashing of the key using SHA3-256. Table 3 further compares our results with elliptic-curve based *ephemeral* key exchange implementations found in [10]. Our implementations also outperform state of the art elliptic curve key exchange protocols by more than a factor of 2, also when considering the scaling due to the more powerful architecture used in this work.

6 Conclusion

We presented constant-time and vector-optimized implementations of NewHope and NewHope-Simple protocols for ARMv8-A 64-bit processors, which exploit NEON extensions together with alternative and more suitable reduction methods for this architecture. Being the first optimized Ring-LWE implementations on this platform, our results show drastic improvements over the C reference implementations and outperform previous results on similar platforms. These results further show the practicability of lattice based post-quantum key exchange along with elliptic-curve cryptography to protect current ARMv8-A based platforms, like mobile phones and tablets.

Availability of Software The software is put in the public domain and available at the web address `https://gitlab.lrz.de/tueisec/NewHope-ARMv8-A`.

References

1. ARMv8-A Architecture. `https://developer.arm.com/products/architecture/a-profile`. Accessed: 2017-02-21.
2. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation, 2016. `http://cryptojedi.org/papers/\#newhopesimple`.
3. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. Document ID: 0462d84a3d34b12b75e8f5e4ca032869, `http://cryptojedi.org/papers/\#newhope`.
4. Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. A new hope on ARM Cortex-M. In Claude Carlet, Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Advanced Cryptography Engineering*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2016 (to appear). Document ID: c7a82d41d39c535fd09ca1b032ebca1b, `http://cryptojedi.org/papers/\#newhopearm`.
5. Yawning Angel. Post-Quantum Secure Hybrid Handshake Based on NewHope. Posting to the tor-dev mailing list, `https://lists.torproject.org/pipermail/tor-dev/2016-May/010896.html`, 2016. Accessed: 2017-04-28.
6. Arch Linux ARM. Arch Linux ARM on ODROID-C2. `https://archlinuxarm.org/platforms/armv8/amlogic/odroid-c2`. Accessed: 2017-02-10.
7. Reza Azarderakhsh, Zhe Liu, Hwajeong Seo, and Howon Kim. NEON PQCryto: Fast and Parallel Ring-LWE Encryption on ARM NEON Architecture. *Cryptology ePrint Archive*, Report 2015/1081, 2015.
8. Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO*, pages 311–323. Springer Nature, 1986.
9. Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, 2008.
10. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. `https://bench.cr.yp.to`. Accessed: 2017-02-21.
11. Daniel J. Bernstein and Peter Schwabe. NEON Crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12*, pages 320–339. Springer, 2012.
12. Ard Biesheuvel. ChaCha20 256-bit cipher algorithm, RFC7539, arm64 NEON functions. `https://git.kernel.org/cgit/linux/kernel/git/ardb/linux.git/tree/arch/arm64/crypto/chacha20-neon-core.S?h=crypto-arm-v4.11`, 2017. Accessed: 2017-02-21.

13. Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem. In *2015 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers (IEEE), may 2015.
14. Jeff Burdges. Post-Quantum Secure Hybrid Handshake Based on NewHope. Posting to the tor-dev mailing list, `https://lists.torproject.org/pipermail/tor-dev/2016-May/010886.html`, 2016. Accessed: 2017-02-21.
15. Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on Post-Quantum Cryptography. Technical report, NIST, apr 2016.
16. Jintai Ding. New cryptographic constructions using generalized learning with errors problem. *Cryptology ePrint Archive*, Report 2012/387, 2012.
17. Jintai Ding, Xiang Xie, and Xiaodong Lin. A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem. *Cryptology ePrint Archive*, Report 2012/688, 2012.
18. Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, NIST, jul 2015.
19. W. M. Gentleman and G. Sande. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference on XX - AFIPS 1966 (Fall)*. Association for Computing Machinery (ACM), 1966.
20. Google. CECPQ1 in BoringSSL. Google Security Blog, `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html`, July 2016. Accessed: 2017-02-21.
21. Shay Gueron and Fabian Schlieker. Speeding up R-LWE Post-quantum Key Exchange. In *Secure IT Systems*, pages 187–198. Springer Nature, 2016.
22. Hardkernel co. ODROID-C2. `http://www.hardkernel.com/main/products/prdt\_info.php?g\_code=G145457216438`. Accessed: 2017-02-10.
23. Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16*, pages 124–139. Springer, 2016.
24. Michael Kerrisk. PERF_EVENT_OPEN Linux Programmer's Manual. `http://man7.org/linux/man-pages/man2/perf\_event\_open.2.html`. Accessed: 2017-02-10.
25. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–519, may 1985.
26. Chris Peikert. *Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, chapter Lattice Cryptography for the Internet, pages 197–219. Springer International Publishing, Cham, 2014.
27. Hwajeong Seo, Zhe Liu, Yasuyuki Nogami, Jongseok Choi, Taehwan Park, and Howon Kim. Parallel Implementation of Number Theoretic Transform. *Cryptology ePrint Archive*, Report 2015/1024, 2015.