A note on the implementation of the Number Theoretic Transform

Michael Scott

MIRACL.com
mike.scott@miracl.com

Abstract. The Number Theoretic Transform (NTT) is the time critical function required by cryptographic protocols based on the Ring Learning With Errors problem (RLWE), a popular choice for post-quantum cryptography. Here we apply a simple methodology to convert the NTT and its inverse from a mathematically correct (but side-channel vulnerable) description, to an efficient constant-time side-channel resistant version.

1 Introduction

Often important cryptographic functions are described in the literature without consideration for side-channel vulnerability. Then they are implemented by competent software engineers who produce functionally correct and efficient real-world implementations, unfortunately without eliminating the side-channels. The result is that many widely used cryptographic libraries contain side-channel weaknesses, that are just awaiting a determined attacker. For a recent example see [4].

What is needed is an intermediate version of the function which is side-channel resistant, from which the engineer can go on to produce their real-world implementation. The purpose of this paper is to describe an easy to follow step-by-step methodology which converts that which is the mathematically correct, to a constant-time side-channel resistant version with the same functionality.

Our particular context is modular arithmetic, that is arithmetic with respect to a prime modulus q. Modular arithmetic is widely used as the basis for many techniques of public key cryptography. Unfortunately it is notoriously hard to implement in constant time, and has been a prime source for side-channel leakage.

In an earlier paper [11] we developed a methodology which we applied to vulnerable functions in the context of elliptic curve cryptography. Here we turn our attention to the post-quantum Ring Learning with Errors (RLWE) method, and its internal NTT transform function (and its inverse).

2 The Methodology

Our starting point is a mathematically correct implementation of the function, which is "exception-free". This is a higher-level requirement for side-channel resistance and for the possibility of a constant time implementation. Basically

it means that at the level above the modular arithmetic there is only a single path through the code, independent of the data it is processing. Fortunately "exception-free" algorithms are commonly available.

Starting with an implementation of such a function, proceed as follows. First identify a modular reduction algorithm that works well with the given prime modulus q. This modular reduction function must operate in constant time, but it is not required to fully reduce its output to be less than q. It is sufficent that the reduction is to a value less that $E.q$, where E is a small positive integer constant. For example the well-known Montgomery modular reduction algorithm reduces its input to a value less than $2q$ [9]. In general it has been observed [7] that such a partial reduction is often easier to achieve than a full reduction, given the constant time constraint. Next

- Associate with every finite field element x an "excess" E_x which tracks the extent to which its value may exceed q in the absence of any reduction, under worst case assumptions. So we know that the unreduced $x < E_x.q$. Input values might be assumed to be fully reduced, in which case they can be initialised to have an excess of 1.
- For modular additions, note that for $z = x + y$, then $E_z = E_x + E_y$, and update and record excesses accordingly.
- Consider all modular subtractions as negation followed by addition, and observe that negation calculated as $-x = E_x.q - x$ will not affect the excess of x.
- For modular multiplications, note that for $z = x.y$, then $E_z = E_x.E_y$ that is the product of the excesses of the inputs, and record a worst-case size of the product $E_z.q^2$ which is to undergo reduction. Set the excess of the output to E.

Next execute the program once. Note that as a consequence of being exception-free, there will be only one path through the code, and so excesses recorded as described above will be invariant irrespective of the actual data being processed.

Now, using these recorded excesses, choose a representation of field elements that (a) would not overflow if modular addition were to be replaced with non-modular addition, and (b) could not under worst case assumptions cause an input to the modular reduction algorithm which would violate its conditions for correct operation. The transcript of the excesses provides a proof that integer overflow cannot occur. Next

- Replace all modular additions, with simple non-modular additions.
- Replace all modular subtractions with negation followed by addition, where $-x = E_x.q - x$
- Perform modular multiplications with a simple non-modular multiplication, followed by our modular reduction algorithm.

Note that for modular negation, it appears to be necessary to continue to track excesses. So as a final step, either replace the excesses with the fixed values

that apply for each individual occurence, or identify a single worst-case value that can be applied in all cases. Assuming this is possible, excesses have now served their purpose, and can be eliminated from the code. We also note that some final post-processing might be required to fully reduce the outputs of the function (if this is required).

3 The NTT

The NTT is basically a form of Discrete Fast Fourier transformation. Once a pair of polynomials is transformed to the "frequency" domain, their product can be calculated by a simple $\mathcal{O}(n)$ element-by-element product, and the result converted back via the inverse transformation (INTT). Therefore the dominant cost of polynomial multiplication, is the cost of the transformation to and from the frequency domain. Hence the significance of the NTT, which has a complexity of just $\mathcal{O}(n \log n)$.

Now when using Fast Fourier methods to determine the product of two polynomials of degree n, this normally requires them to be first padded to length $2n$. However in the RLWE setting we get around this problem by manipulating our polynomials modulo $x^n + 1$, and using the negative wrapped convolution.

The fixed system parameters for a typical instance of the Ring Learning with Errors problem consist of a prime modulus q and polynomials of degree n where $n = 2^m$, and with coefficients $\in \mathbb{F}_q$. In most cases the prime modulus q is selected such that $q = 1 \bmod 2n$, so that the $2n$-th roots of unity exist and can be precalculated. Typical choices might be $q = 12289$, and $n = 1024$, as used in the NewHope proposal [2], in which case a 2048-th root of unity would be 9089, and it is easily confirmed that $9089^{2048} = 1 \bmod 12289$. To implement the NTT we will need to precompute a vector of the first n powers of such a root, but stored in bit reverse order. So given a root g, while the natural ordering of its powers would be $[g^0, g^1, g^2, \ldots g^{n-1}]$, in bit reverse order the element g^i would actually be stored at an index in the table found by reversing the m bits in i.

Our starting point is the basic NTT algorithm and its inverse, based on its description by Naehrig and Longa [8], which integrates many prior optimizations. A key idea is to use the Cooley Tukey butterfly for the forward transformation, but to switch to the Gentleman-Sande butterfly for the inverse operation. See Algorithms 1 and 2.

Our ultimate aim is to eliminate the side channel leakage from the modular arithmetic, that is from those clearly indicated calculations that take place modulo q. One clever optimization that is not shown here, but which we will make use of in all of our implementations, is to merge the last iteration of the main INTT loop with the multiplication of each element of its output by $n^{-1} \bmod q$, which saves $n/2$ modular multiplications. See [8] for details.

Algorithm 1 The Cooley-Tukey NTT algorithm

INPUT: A vector $\boldsymbol{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$

INPUT: Precomputed table of $2n$-th roots of unity $\boldsymbol{g}$, in bit reversed order

OUTPUT: $\boldsymbol{x} \leftarrow NTT(\boldsymbol{x})$

```
1:  function NTT(x)
2:      t ← n/2
3:      m ← 1
4:      while m < n do
5:          k ← 0
6:          for i ← 0; i < m; i ← i + 1 do
7:              S ← g[m + i]
8:              for j ← k; j < k + t; j ← j + 1 do
9:                  U ← x[j]
10:                 V ← x[j + t].S mod q
11:                 x[j] ← U + V mod q
12:                 x[j + t] ← U − V mod q
13:             k ← k + 2t
14:         t ← t/2
15:         m ← 2m
16:     return
```

Algorithm 2 The Gentleman-Sande inverse INTT algorithm

INPUT: A vector $\boldsymbol{x} = [x_0, \ldots, x_{n-1}]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus $q = 1 \bmod 2n$

INPUT: Precomputed table of inverses of $2n$-th roots of unity $\boldsymbol{g^{-1}}$, in bit reversed order

INPUT: $n^{-1} \bmod q$

OUTPUT: $\boldsymbol{x} \leftarrow INTT(\boldsymbol{x})$

```
1:  function INTT(x)
2:      t ← 1
3:      m ← n/2
4:      while m > 0 do
5:          k ← 0
6:          for i ← 0; i < m; i ← i + 1 do
7:              S ← g^{-1}[m + i]
8:              for j ← k; j < k + t; j ← j + 1 do
9:                  U ← x[j]
10:                 V ← x[j + t]
11:                 x[j] ← U + V mod q
12:                 W ← U − V mod q
13:                 x[j + t] ← W.S mod q
14:             k ← k + 2t
15:         t ← 2t
16:         m ← m/2
17:     for i ← 0; i < n; i ← i + 1 do
18:         x[i] ← x[i].n^{-1} mod q
19:     return
```

3.1 A Naïve solution

An initial reaction might be that surely we could simply replace modulo q everywhere with $\%q$, and let the processor's built in integer division/remainder instruction take care of it.

But this is a very unsatisfactory solution for a number of reasons. Firstly integer remaindering is not the same as modular reduction, and commonly requires conditional corrections to keep results in range. More importantly integer division is complex to implement in hardware, is rarely a bottleneck calculation in most computer applications, and therefore is very slow, and of particular relevance to us, most often not implemented in constant time [5]. That is takes a number of clock cycles that is dependent on the data being processed. Nevertheless this makes a good starting point from which to calculate and record excesses, and to develop better solutions.

We focus on the "butterfly" code in the innermost loops of the NTT and INTT algorithms. Once we move from an algorithmic description to actual code, we need to become aware of the possibility of integer overflow. We will assume two signed integer data types, which we will refer to as `int_t` and `int_dt` (wordlength and double-wordlength), where the actual wordlength `WL` might be 16, 32 or 64 bits. In the C programming language on a 32-bit computer these types might be `int32_t` and `int64_t` respectively. We refer to their unsigned equivalents as `uint_t` and `uint_dt`. For a particular implementation we assume that the modulus q is globally visible.

Of course polynomials might be transported and stored using a smaller data type, if the modulus q is small. For example the commonly suggested 14-bit prime 12289 will fit comfortably in a 16-bit type irrespective of the wordlength of the processor.

Listing 1.1: Naïve Modular multiplication

```
int_t modmul(int_t a,int_t b)
{
        return (int_t)(((int_dt)a*b)%q);
}
```

Listing 1.2: Naïve method for NTT

```
U=x[j];
V=modmul(x[j+t],S);
x[j]=(U+V)%q;
x[j+t]=(U+q-V)%q;
```

Listing 1.3: Naïve method for INTT

```
U=x[j];
V=x[j+t];
x[j]=(U+V)%q;
W=(U+q-V);
x[j+t]=modmul(W,S);
```

It becomes immediately obvious that our ability of optimize the code will depend on the extent of the allowable excess that exists, which will allow intermediate values in the butterfly computation to increase outside of the rigid range $0 \rightarrow q$, without overflow. For the above code to function correctly it is already assumed that calculating $U+V$ in listing 1.3 does not cause an overflow. We will initially assume a signed type of a length at least 2 bits bigger than the prime modulus. Making immediate use of this latitude, observe that in the INTT case we can calculate W without reduction modulo q.

3.2 A Constant Time solution

However this is not a constant-time solution, and its slow. We need an alternative technique, and Montgomery's method for modular multiplication without division [9], which replaces division by some multiplications, is ideal, widely used in this context, and generically applicable. Note that integer multiplication, unlike division, is heavily optimized for most processors, and usually executes in a fixed number of clock cycles, often just 1 or 2. So we can anticipate that such an implementation might even be faster. One downside to Montgomery's method is that we need to convert field elements to and from Montgomery representation before the NTT and after INTT functions. An obvious optimization is to precompute the tables of roots of unity and their inverses in Montgomery format.

The Montgomery method assumes the choice of an alternate modulus R greater than q, which is a simple power of 2, the idea being to replace the modulo q calculation with a much simpler reduction modulo R. For maximum efficiency it is common to choose R to be 2 to the power of the wordlength, and this is what we will use here. The method also requires the precomputed constant $N = 1/(R-q) \bmod R$.

Montgomery's method introduces a modular reduction function `redc` which reduces an input T to a positive integer t less than $2q$, assuming that $T < qR$. Therefore reduction is not complete. However a simple constant-time augmentation can complete the reduction. Subtract q from t and do an arithmetic shift right by one less than the bitlength of the signed type used for t. If $t-q < 0$ this results in all ones, otherwise all zeros. Perform a logical AND of this bit pattern with q and add it back into t.

Note the condition on the input to `redc` that it be less than qR. Since in this context one of the inputs is from a precomputed table, and hence less than q, it merely suffices that the other is less than R and is representable as an `int_t`.

The conversion of x to Montgomery form can be computed by applying the `redc` function to the product of x and the precomputed constant $R^2 \bmod q$. The conversion back to "normal" form is simply an application of `redc` [9].

Listing 1.4: redc function with full reduction

```
int_t redc(int_dt T)
{
    uint_t m=(uint_t)T*N;
    int_t V=(((uint_dt)m*q+T)>>WL);
    V-=q; V+=(V>>(WL-1))&q;
    return V;
}
```

Listing 1.5: Modular multiplication

```
int_t modmul(int_t a,int_t b)
{
    return redc((int_dt)a*b);
}
```

Listing 1.6: Constant Time method NTT

```
        U=x[j];
        V=modmul(x[j+t],S);
        W=U+V-q;
        x[j]=W+((W>>(WL-1))&q);
        W=U-V;
        x[j+t]=W+((W>>(WL-1))&q);
```

Listing 1.7: Constant Time method INTT

```
        U=x[j];
        V=x[j+t];
        W=U+V-q;
        x[j]=W+((W>>(WL-1))&q);
        W=U+q-V;
        x[j+t]=modmul(W,S);
```

3.3 Lazy Reduction

Next we apply our proposed methodology to make maximum use of delayed reductions. Ideally we will succeed in removing all of the reduction code, other than that implicit in the basic un-augmented `redc` function. We would expect our ability to achieve this to depend on the detail of the NTT and INTT algorithms, and on the number of excess bits available to us, which will facilitate delayed reduction.

We make a minor change from the description of our methodology above. Since our conversion to Montgomery form now uses the un-augmented `redc` function, we will assume initial excesses of 2 rather than 1. However we still assume that precomputed values such as the roots of unity are fully reduced.

One immediate and striking observation is that the Cooley-Tukey NTT and Gentleman-Sande INTT butterflies behave very differently. As the iterations progress the excesses get bigger. But whereas the Cooley-Tukey excesses increase only slowly and linearly, the Gentleman-Sande worst case excesses grow much more rapidly.

First examine the Cooley-Tukey butterfly. Observe that x values are incremented by the output of the a modular multiplication, which will have a maximum excess of 2. So the new excesses will be at most 2 bigger than an existing excess. But for the Gentleman-Sande butterfly certain x values may have their excesses doubled by the execution of the code `x[j]=x[j]+x[j+t]`. And this is what we observe.

Recall that there are two places where integer overflow might occur due to excessive excesses, after addition and before modular multiplication. Experimentally we determine that, for polynomials of degree n, for the NTT (based on Cooley-Tukey) the maximum excess is $2.\lg n + 2$, and for the INTT (based on Gentleman-Sande) the maximum excess is $2n$.

Assuming for the moment that these excesses can be accomodated, we can replace the constant time code with the following. Note that all explicit modular reduction code has been removed. In the case of INTT the worst-case excess for V, which is just the polynomial degree n, is used in the calculation of W in listing 1.10, and so explicit use of the excesses is not required. Obviously $2q$ and nq can be precalculated.

Listing 1.8: redc function incomplete reduction

```
int_t redc(int_dt T)
{
        uint_t m=(uint_t)T*N;
        int_t V=(int_t)(((uint_dt)m*q+T)>>WL);
        return V;
}
```

Listing 1.9: Lazy Reduction method for NTT

```
        U=x[j];
        V=modmul(x[j+t],S);
        x[j]=U+V;
        x[j+t]=U+2*q-V;
```

Listing 1.10: Lazy Reduction method for INTT

```
        U=x[j];
        V=x[j+t];
        x[j]=U+V;
        W=U+n*q-V;
        x[j+t]=modmul(W,S);
```

Finally we need to consider the conditions under which the excesses that might arise, can be safely accomodated. Since the worst case arises for the INTT

butterfly, this code will work correctly as long as $2nq$ can be represented in the `int_t` type. For example on a 32-bit processor, the C language type `int32_t` can comfortably handle the case of $q = 12289$ and $n = 1024$. However for larger values of q we can expect problems to arise. For example consider the parameters chosen by Güneysu et al [6], where $q = 8383489$ and $n = 512$. This will cause a problem for our INTT code on a 32-bit processor.

However from our analysis we know exactly where the worst case excesses occur, and so we can compensate for it. By inserting extra reduction code at the appropriate point in the INTT function, it has the effect of suppressing the excesses. Note that reduction of any value can be achieved at any time by multiplying it by the Montgomery representation of unity ($O = R \bmod q$). When extra reductions are introduced, the excess transcript can be examined to determine whether or not the correction has succeeded.

Experimentally we have determined that the modified code in listing 1.11 seems to work well. Set L as the smallest power of 2 such that $2(n/L)q < 2^{31}$. For $q = 8383489$, and $n = 512$, then $L = 4$. Since the corrections are only rarely required, the performance impact should be small. However we appreciate that such measures will eventually become less effective as q increases, and the available excess diminishes.

Listing 1.11: Modified Lazy Reduction method for INTT

```
        if (m<L && j<k+(L/2*m))
        {
                U=modmul(x[j],O);
                V=modmul(x[j+t],O);
        }
        else
        {
                U=x[j];
                V=x[j+t];
        }
        x[j]=U+V;
        W=U+(n/L)*q-V;
        x[j+t]=modmul(W,S);
```

3.4 Special Moduli

Special form moduli can be used and exploited in our framework as long as they obey the same rules as Montgomery arithmetic, that is for an input $< qR$ they produce an output less than $2q$. For example the Fermat prime $2^{16} + 1 = 65537$ which has been proposed for RLWE implementations [10], has a fast reduction, and does not require field elements to be converted to and from Montgomery form, with further savings. See listing 1.12 for the fast reduction code for a 32-bit processor. Careful analysis confirms that the output will always be positive and less than $2q$.

Listing 1.12: reduction function for Fermat prime 65537

```
int32_t redc(int64_t T)
{
        T=(uint_t)T+(T>>32);
        return (T&0xFFFF)+q-(T>>16);
}
```

4 Mapping implementations to platforms

Our näive and constant time implementations will work immediately on a 16-bit processor, where a `int_t` is represented by a 16-bit C type like `int16_t`, and `int_dt` maps to a `int32_t`, assuming that the prime modulus q is 14-bits or less. Unfortunately in this setting the available excesses are insufficient for our full lazy reduction approach. But in many cases q is bigger than 16-bits, although usually less than 32-bits, in which case a 32-bit (or 64-bit) processor is really a necessity, where `int_t` maps to `int32_t`, and `int_dt` maps to `int64_t`. The majority of primes suggested for RLWE range from 13 to 26 bits [1]. In these cases our lazy reduction code will be a good fit on a 32-bit processor.

5 Comparison with prior art

In their influential paper Alkim et al [2] provide a reference C implementation of the NTT, using the Gentleman-Sande approach. Their solution is closer to our constant time solution, and appears to be targeted at a 32-bit architecture, but one without a 32x32 multiplier. Such architectures exist, and a prime example would be the ARM Cortex-M processor, which they specifically targetted in a follow-up paper [3]. By using the 14-bit prime $q = 12289$ and a Montgomery modulus of 2^{18}, they cleverly succeed in squeezing the arithmetic into 32-bits (as 14+18=32). As $R = 2^{18}$ is a few bits greater than q a modest amount of lazy reduction then becomes possible.

The paper by Longa and Naehrig [8], which was the starting point for this research, ends up with an implementation not very dissimilar to our own, albeit they come to it by a different route. The main difference is that they choose to use a modular reduction method tailored to the specific types of primes used in RLWE, that is primes such that $q = 1 \bmod 2n$. Therefore they do not use Montgomery reduction, but can hence avoid the transformation to/from Montgomery form, with further savings. Their solution is appropriate to a more conventional 32-bit architecture which allows 64-bit products.

However the Longa and Naehrig implementation is described only in the context of a particular choice of parameters, namely $q = 12289$ and $n = 1024$, as used for the NewHope key exchange protocol described in [2]. Their implementation, like ours, requires extra modular reductions introduced at certain steps in the NTT algorithm and its inverse. The reasoning for the positioning of these extra reductions is not fully explained, and hence it is not clear when they would be required for a different choice of parameters. However we can see now that they are introduced as a mechanism to suppress the excesses from getting too large. Using their special reduction function, it appears that the extra reductions must be introduced into both the NTT and the INTT code, whereas in our implementation our analysis shows that they are not needed at all for the NewHope parameter set, and are only required for the INTT code when q gets much larger.

6 Results

Our code is available here [1].

First we provide some comparative timings, using cycle counts obtained from an Odroid C2 single board computer, as used by Streit and De Santis [12] in their implementation of NewHope on an ARM Cortex-A53 processor. Following their example, we obtain hardware cycle counts using the accurate Linux Kernel performance monitoring system call, using the GCC compiler version 5.4 with maximum optimization. We provide results for all three methods described here, for the parameters $q = 12289, n = 1024$ and $q = 8383489, n = 512$, and $q = 16760833, n = 1024$ [1] to demonstrate that our code is not tied to just one fixed set of parameters. See table 1.

Prime q	Degree n	Method	NTT	INTT
12289	1024	Naive	161701	127879
12289	1024	Constant Time	102306	91223
12289	1024	Lazy Reduction	74174	78108
8383489	512	Naive	96369	74126
8383489	512	Constant Time	48478	43273
8383489	512	Lazy Reduction	35537	39784
16760833	1024	Naive	196473	133014
16760833	1024	Constant Time	102208	91059
16760833	1024	Lazy Reduction	74206	84088

Table 1: Odroid C2 Cycle counts

For the NewHope parameters we note that our cycle counts are nearly exactly half of those quoted by Streit and De Santis, who used the C reference code from [2]. Using our new counts, it would appear that the advantage of using NEON instructions is not the speed-up of 8.3 as claimed, but is closer to (a still very impressive) 4. We next adapted the Longa and Naehrig code to use the same performance counters, and observed that for the NewHope parameters their code is about as fast as ours (79020 for NTT, 75822 for INTT). However we would contend that our reference code is much more general purpose.

Next we performed the same measurements this time using an Intel i5-6400 processor with Turbo Boost disabled, and the GCC version 5.3 compiler.

Prime q	Degree n	Method	NTT	INTT
12289	1024	Naive	70327	46997
12289	1024	Constant Time	47017	42424
12289	1024	Lazy Reduction	34300	35110
8383489	512	Naive	37455	23350
8383489	512	Constant Time	21891	19587
8383489	512	Lazy Reduction	15863	16703
16760833	1024	Naive	71800	51446
16760833	1024	Constant Time	47392	43160
16760833	1024	Lazy Reduction	35165	37655

Table 2: Intel i5 Cycle counts

[1] `indigo.ie/~mscott/ntt_ref.c`

In this case the Longa and Naehrig code when measured on our compiler/processor combination, was about 5% faster than our lazy reduction code for the NewHope parameters.

We observe that the extra reductions necessary for the INTT code to work correctly for the larger primes 8383489 and 16760833, do not appear to significantly effect the performance.

7 Conclusion

We have described an improved reference C implementation of the Number Theoretic Transform and its inverse, as required for the implementation of post-quantum cryptographic schemes based on the Ring Learning With Errors problem. The implementation is efficient and constant time, and hence a safe starting point for more highly optimized code. It can be used with a range of parameters, and is easily translated to other languages. Our solution uses a methodology which allows the idea of Lazy Reduction to be exploited to the full, with confidence that integer overflow will never occur. Our methodology exposes the surprising observation that the Cooley-Tukey butterfly is much more lazy-reduction-friendly than the Gentleman-Sande alternative.

Another conclusion from our results is that, if one ignores bandwidth considerations, the cost of moving to a larger value for q has only a negligible impact on performance.

Finally we conclude that some performance improvements claimed for assembly code that exploits instruction set extensions like Intel AVX2 and ARM NEON, when compared to compiler-generated C code, while still very impressive, are perhaps not quite as good as they seemed.

References

1. Ring learning with errors parameters, 2017. `http://www.ringlwe.info/parameters-for-rlwe.html`.
2. E. Alkim, L. Ducas, T. Poppelmann, and P. Schwabe. Post-quantum key exchange – a new hope. In *25th Usenix Security Symposium*, pages 327–343, 2016.
3. E. Alkim, P. Jakubeit, and P. Schwabe. New Hope on ARM cortex-M. In *SPACE 2016*, volume 10076 of *Lecture Notes in Computer Science*, pages 332–352. Springer, 2016.
4. D. Bernstein, J. Breitner, D. Genkin, L. Groot Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom. Sliding right into disaster: Left-to-right sliding windows leak. Cryptology ePrint Archive, Report 2017/627, 2017. `http://eprint.iacr.org/2017/627`.
5. A. Fog. Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs, 2017. `http://www.agner.org/optimize/`.
6. T. Güneysu, T. Oder, T. Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *PQCrypto 2013*, volume 7932 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2013.
7. D. Harvey. Faster arithmetic for number-theoretic transforms. *J. Symb. Comput.*, 60:113–119, 2014.
8. P. Longa and M. Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *CANS 2016*, volume 10052 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2016.
9. P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
10. T. Pöppelmann and T. Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *LatinCrypt 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2012.
11. M. Scott. Slothful reduction. Cryptology ePrint Archive, Report 2017/437, 2017. `http://eprint.iacr.org/2017/437`.
12. S. Streit and F. De Santis. Post-quantum key exchange on ARMv8-A – a new hope for NEON made simple. Cryptology ePrint Archive, Report 2017/388, 2017. `http://eprint.iacr.org/2017/388`.