Performance Evaluation and Optimisation for Kyber on the MULTOS IoT Trust-Anchor

Keith Mayes
Information Security Group
Royal Holloway, University of London
Egham, UK
keith.mayes@rhul.ac.uk

***Abstract*—The Internet of Things (IoT) may be considered as a distributed, critical infrastructure, consisting of billions of devices, many of which having limited processing capability. However, the security of IoT must not be compromised by these limitations, and defenses need to protect against today's threats, and those predicted for the future. This requires protection against implementation attacks, as well as the ability to load, replace and run, best-practice cryptographic algorithms. Post-Quantum cryptographic algorithms are attracting great interest, and NIST standardization has a competition to find the best. Prior research demonstrated that a Learning With Errors candidate algorithm could be implemented on a smart card chip, however this was a low-level implementation, and not representative of loading the algorithm onto a secured IoT chip platform. In this paper we present analysis from a practical implementation of the Kyber768 CPAPKE public key encryption component on a MULTOS IoT Trust-Anchor chip. The investigation considered memory and speed requirements, and optimizations, and compared the NTT transform version of Kyber, presented in Round 1 of the NIST competition, with the Kroenecker multiplier technique that exploits a hardware crypto-coprocessor. The work began with a generic multi-round multiplier approach, which was then improved using a novel modification of the input data, allowing a built-in modular multiply function to be used, significantly increasing the speed of a multiplication round, and doubling the useable size of the hardware multiplier.**

***Index Terms*—MULTOS, Kyber, Post Quantum, embedded, performance, IoT**

I. Introduction

The Internet of Things (IoT), is fast evolving into a critical enabler for future society. Much focus is on new functionality and services, however ensuring the security of IoT is crucial. As yet there is no clear solution for securing the entire IoT, however a lot is known about providing system security in legacy systems, using best-practice cryptographic algorithms and protocols, and there is considerable industry expertise in protecting security sensitive devices from attacks on their implementation. Complications for IoT security, include the long potential life of the deployed infrastructure, the difficulty to physically access and/or replace security-sensitive devices, and their processing resource limitations. To maintain an effective defense against evolving attacks, requires flexible security devices, which even after deployment, can be loaded with new algorithms. The greatest test for such devices, may eventually be attackers equipped with quantum computers, implying that we cannot rely on legacy Public Key Infrastructure (PKI) for confidentiality, integrity and availability.

In this paper we do not offer a magic bullet for IoT security, but, practically investigate, trustworthy security foundations for IoT that support traditional algorithms, yet are sufficiently flexible to support the future loading of post-quantum algorithms. The considered scenario was the IoT seeded with post-quantum capable security anchors.

There are numerous security-sensitive systems in use today, which have protection from strongly attack-resistant hardware, e.g., the chips in our bank cards and passports. They include specialist hardware, to resist physical, side-channel and fault attacks (summarized in [11]), supported by software defensive measures, and are typically assessed under Common Criteria (CC) [2]. The secured microcontrollers within bank cards, are normally of small register size (16-bit is common) and have limited memory and processing speed; the Infineon SLE78 [7] is typical. The software defensive measures for high level CC evaluation, significantly degrade performance compared to unprotected native mode implementations, with one or two orders of magnitude not untypical (see [10]). Therefore, a secured microcontroller has a crypto-co-processor (CCoP), with special hardware for executing specific functions much faster than the CPU. The functions may be complete algorithms, e.g., RSA [15], or general utilities such as block multiplies. A secure chip *platform* will offer an API for functions that map onto the underlying CCoP. The CCoP cryptographic operations may be fast relative to simple byte or bit manipulation via the main CPU; making results and optimisation strategies unusual compared to a CPU without CCoP.

The MULTOS [13] platform has a CCoP, but also offers generic software primitives that still have defensive coding, but are optimized for faster execution compared to implementation at the application level. Secure chips are typically initialized and personalized before first use, which may include the storage of identities and cryptographic keys for operation and management. To overcome processing restrictions, some values may be pre-computed, for example, storing a diversified ID rather than calculating it, or adding small look-up tables to speed execution; our work made extensive use of the personalisation phase. In this research we chose to use MULTOS security platform(s) based on the high-levels of security assurance that they have achieved, and the availability

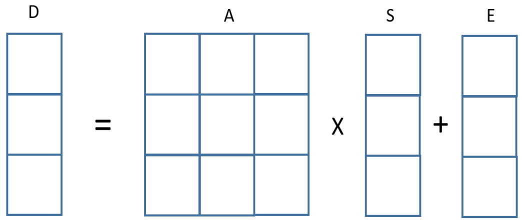

Fig. 1: Kyber Calculation

of the new IoT Trust-Anchor device. For the algorithm we chose Kyber768 [16] CPAPKE, primarily because it was the subject of the main prior-research reference, but also because the inherent polynomial multiplications would stress our implementation.

In Section II we provide a brief overview of Kyber and then in Section III we outline initial assumptions and strategy for the implementation. Section IV introduces the essential aspects of MULTOS. In Section V we describe the implementation process and challenges. Section VI presents, analyses and optimizes the Part I experimental results; initially comparing the NTT and generic multiplier methods, and then going on to analyse the timing breakdown of the latter. In Section VII the Part II work and results are discussed, which are based on a novel multiplication strategy, possible when one of the inputs represents noise. Section VIII offers conclusions and suggests future research.

II. Kyber

Kyber is a Key Encapsulation Mechanism, within the NIST competition [14] for standardising post-quantum algorithms; as an alternative to well-known asymmetric algorithms, such as RSA. There are different classes of post-quantum algorithms, and particular algorithm proposals are evolving and combining as the competition progresses. The theoretical security properties of Kyber are left to the international community of cryptographic experts, as we focus on its implementation, as a downloaded algorithm on an attack-resistant hardware platform. The security of Kyber is based on the hardness of the Module Learning With Errors problem (MLWE). This is an algebraic assumption part of the larger family of Learning With Errors (LWE), for which an introduction can be found in [17]. The design of Kyber as a PKE scheme follows that outlined in [9]. In terms of basic encryption process, there is some population of fields with random values, multiplication with a generator matrix and a fixed key vector, and then the intentional combination with a noise vector; as illustrated in Figure 1. The security design goal is to make it infeasibly difficult for an attacker to determine either the private key S or the random noise vector E from a knowledge of the result D and the generator matrix A. Note that we also use A transpose (A^T) in calculations, but we will just refer to A for simplicity of explanation.

The calculations centre around polynomials. A polynomial has 256 coefficients, with each coefficient being represented as a modulo KYBER_Q (7681) unsigned integer. In the Kyber reference implementation, and in this work, each coefficient is stored within a 2-byte/16-bit unsigned integer. A polynomial data structure is referred to as a ”*poly*” in the reference code, and every small box in Figure 1 represents a *poly*. The secret key S and noise E are vectors containing three *poly* types; and referred to as ”*polyvecs*”. The generator matrix A consists of three *polyvecs*. One of the most time-critical operations within the algorithm, is the multiplication of *polyvecs*; which for the multiplication of A by S is required three times. Each *polyvec* multiplication consists of three *poly* multiplications, and subsequent addition of their results. In the original specification of Kyber, the *polyvecs* are encoded in the Number Theoretic Transform (NTT) domain, reducing classical polynomial multiplication to vector inner products. Note that as any A matrix would transform into another A matrix, the original Kyber was able to avoid the transform, by stating that the generated matrix was already transformed. The paper [1] swapped the transform approach for the use of a hardware multiplier and we will follow this strategy; although an NTT version was ported to the test platform for comparison. We note that there is considerable research activity to determine specialist hardware implementations for NTT (and LWE) such as in [3], [6], which contrasts with our goal of using an existing and flexible attack-resistant platform, with generic CCoP support. For clarity we will recap here on the multiplier method.

A. Multiplier-Method

Polynomial multiplications are at the heart of Kyber, and critical in our use of the MULTOS Trust-Anchor, but such operations can be slow without specialist hardware. The Kyber reference uses the NTT transform as a means to avoid costly polynomial multiplication, however there is a time cost for the transform and its inverse. The method proposed in [1] avoids the transform cost, by exploiting a hardware multiplier within the CCoP. It turns a polynominal multiplication into large integer multiplications. The CCoP hardware multiplier is not as large as the multiplication required by Kyber. Therefore, the multiplications have to be broken into interleaved parts, with each handling a sub-set of coefficients, and then an accumulation of the interleaved results. For each sub-multiply (we refer to as Rounds), we use the Kronecker substitution [8] as per the prior research [1]. The principle is to space the polynomial coefficients sufficiently far apart within the large input integer, that they do not overflow their storage as a result of the multiplication, so the output coefficients can be simply extracted from the result. For Kyber, our input coefficients are mod Kyber_Q (7681), so 7680 is the largest magnitude, requiring a minimum 13-bit storage location, and 26-bits for the product of two coefficients. In our initial implementation, for each round of multiplication, we had 32 (of the 256) polynomial coefficients within each long integer input, meaning that an unreduced resulting coefficient could be the sum of 32 of the 26-bit products; requiring a total of 31-bits of storage and fitting within the 32-bits provided. Note that for the initial implementation, the full result had to be available from the multiplier, in order to support a modular reduction, whereby the upper (overflow)coefficients were subtracted (modulo Kyber_Q) from the lower (retained) coefficients. This is discussed further in Section V-B, but accounts for why we could only multiply 32 coefficients at a time (within a 2048-bit multiplier). In our final implementation we succeeded in multiplying 64 coefficients at a time, within the same size multiplier; as discussed in Section VII.

III. Implementation, Initial Assumptions and Strategy

The focus of the research was to determine whether the Kyber768 algorithm, as an example of a polynomial based algorithm, could be loaded as an application onto a MULTOS VM platform and execute acceptably via its APIs. As the critical aspects centred around *polyvec* multiplication, a scenario was envisaged in which a test message was encrypted and decrypted using keys/generators known in advance and personalized on the chip. It was important to prove, that the MULTOS code was correctly matching that of the reference; which would not have been possible if using MULTOS random numbers, or different hash types. To achieve this, the Linux reference code was used to dump its ”random” data fields, as well as data representing generators, keys and noise; which were subsequently personalized into the Non-Volatile-Memory (NVM) of the MULTOS test device. Additional data was pre-computed to support the multiplier method; for example, the inverse NTT version of the original matrix *A* was needed.

IV. MULTOS

Our MULTOS device can be considered as a secure Virtual Machine (VM) running on an attack-resistant chip, plus external systems and processes that manage personalisation of the chip with cryptographically protected Application Load Units. The management and personalisation are currently based on PKI, but would eventually evolve to a post-quantum equivalent. The PKI loading approach is interesting for IoT, as the distribution of the more commonly-used symmetric keys does not scale well. The security evaluated MULTOS management capability, provides reasonable justification that the chips can be securely loaded and personalized post-deployment. As would be expected from a secure multi-application device; loaded applications are isolated from each other and under the unique control of the Issuer. Source code development for MULTOS is normally in the ”C” language, which is compiled to MULTOS Execution Language (MEL). MEL can be regarded as a kind of assembly language, however, for security, it is greatly abstracted from the underlying processor, and instructions are better thought of as functions. The VM is 16-bit, with 32-bit extensions for memory access, and the compiler supports up to 32-bit integer types. MULTOS applications run within a virtual address memory map, which is shown in Figure 2.

The main memory split is between code and data spaces, both of which can be addressed up to 64kbyes. Historically, the code (and static data) spaces, were held in ROM, with no option to change the contents during deployment. Nowadays, flash NVM is used, with the code being logically read-only, but with the possibility of overwriting static data. The NVM is fast to read, yet very slow to write; so useful for personalized data, but not for dynamic variable storage. Run-time data and variables are in RAM; some is needed for the stack to support the OS as well as loaded applications. The application also needs private RAM (session data) for variables, buffers, arrays etc.. There is some public data, used for I/O buffers and communicating between applications; which can (with some care) be re-used as extended session memory; the motivation being that the RAM for the entire chip is usually a small fraction of the maximum 64kbytes that MULTOS would support within one application’s data space.

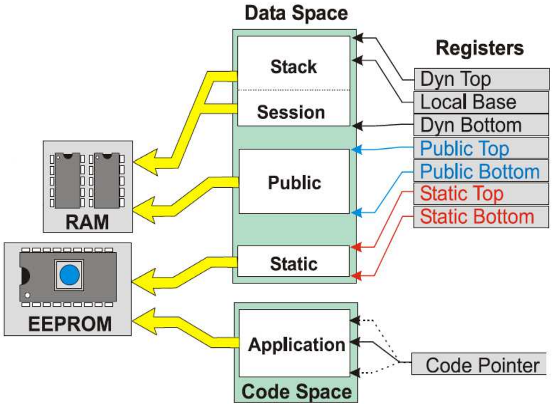

Fig. 2: MULTOS Application Memory Architecture

A. Memory requirements

Generally, application coding is unlikely to exceed the 64kbyte code memory space, so the focus was on the RAM and to a lesser extent on the data NVM. The latter was only relevant due to the pre-calculated data needed for the experiments. In the core calculation, we need to multiply a *polyvec* by another; to have a generator matrix and at least another *polyvec* stored in NVM. The storage of a normal *poly* is 256*2 = 512 bytes. A *polyvec* contains three *polys* and therefore needs 1,536 bytes, and we need two. We also need buffers to expand the *polys* for the Kronecker substitution, with four bytes per coefficient, so 1024 bytes each, and we need two. Allowing for other buffers and variables, but assuming some re-use of the *poly/polyvec* memory, we estimated a minimum of 6kbytes application RAM, plus enough to support the OS and the application stack requirements. For the static data, the largest entry would be the generator matrix *A*, which is three *polyvecs*, and so 4608 bytes; allowing for a similar number of individual *polyvecs* we would need 9216 bytes, and reserving half again for static values and look-up tables we are in the range of 14kbyes. Fortunately, the memory sizes fit within the logical limits of the MULTOS virtual memory. The static memory is no concern, as MULTOS chips usually have a few 100kbytes of NVM. The RAM requirement is just within the MULTOS Trust-Anchor [12], which has a total of 13kbytes RAM; 9.5kbytes for stack and session, and 3.5kbytes for public data.

B. Hardware Multiplier API

The Trust-Anchor has a CCoP, although at 2048bits, it is half the size of the one used in [1]. The MULTOS API supports two ways to access the multiplier; the simplest being *multosMultiply*, which mapped to the *MULTIPLYN* primitive.

In our Part I implementation, we multiply inputs of 32 coefficients, each stored in 4-byte storage, equivalent to 128-byte large integer inputs. In theory, *multosMultiply* should have been capable of multiplying two 128 byte unsigned integers and producing a 256 byte result, however the *STORE* primitive is restricted to a maximum length of 255 bytes, whereas *blockLength*2* is 256 bytes. In any case, the *multosModularMultiplication* was more interesting, as when the Kronecker multiply is broken down into sub-multiplies, it is necessary to do a modular reduction after each, to get back to a 32 coefficient result. As will be explained later, for Part I of the study, *multosModularMultiplication* could not be used to exactly duplicate the NTT reference results. However, despite the input parameters and modulus having to be the same size, we were still able to configure*multosModularMultiplication* to be a reliable functional equivalent of *multosMultiply*, but we were then obliged to carry out the reduction manually in application level software.

V. Software Implementation

The software starting point was the NIST Round 1 reference version from GiTHub [5]. This was built on a Linux machine and modified so that data values could be dumped from the code for import as pre-personalized data within the MULTOS test-code. The exported data included the transposed version of the generator matrix A^T used in the NTT calculation, as well as a version of this matrix calculated by inverse NTT transform on A^T; required for the multiply version. The unpacked public key was stored in a *polyvec*, *pkpv*. To be consistent with the NTT version, an NTT transformed version of *pkpv* was also pre-stored, and the NTT transformed version of the unpacked secret key *sk*, as well as an untransformed version. Noise vectors *sp, ep* and *epp* were also stored to avoid any difference from sources of randomisation. The initial focus of the investigation was around the performance of *indcpa_enc* and *indcpa_dec* functions; exploiting pre-stored values and comparing the NTT approach with the multiplier equivalent.

A. NTT and Multiplier Version

After using pre-personalized keys and data fields where possible, the essential part of *indcpa_enc* is outlined in Listing 1, along with the multiplier equivalent. Note that KYBER_K is '3', and the original performance critical parts were the NTT transform, *polyvec_ntt*, the inverse transforms, *polyvec_invntt* and *polyvec_pointwise_acc*; the latter being much simplified by the transforms. The NTT version was primarily to show exact functional equivalence, but its timing was also measured as an initial benchmark. The experimental timing is shown in Table I. As anticipated, the overall duration was much too slow to be practical on our VM, although the breakdown of the timing was encouraging; much of the execution time was spent in transforms and inverses, which are not needed in the multiplier approach

```
NTT  ORIGINAL                            MULTIPLIER
poly_frommsg(...);                  poly_frommsg(...);
polyvec_ntt(...);                   ...
for(i=0;i<KYBER_K;i++)              for(i=0;i<KYBER_K;i++)
  polyvec_pointwise_acc(...);         polyvec_multiply_acc(...);
polyvec_invntt(...);                ...
polyvec_add(...);                   polyvec_add(...);
polyvec_pointwise_acc(...);         polyvec_multiply_acc(...);
poly_invntt(...);                   ...
poly_add(...);                      poly_add(...);
poly_add(...);                      poly_add(...);
pack_ciphertext(...);               pack_ciphertext(...);
```

Listing 1: Overview of Encrypt Methods

The *poly_frommmsg*, the last two *poly_adds* and the *pack_ciphertext* are common to both versions. The multiplier performance is dominated by *polyvec_multiply_acc*, which has three calls to *polymodmul* (which multiplies two polynomials together), after which a process adds the three sets of coefficient results and reduces them mod KYBER_Q. Note that *polyvec_multiply_acc* is called four times in *indcpa_enc*, so *polymodmul* is called 12 times; becoming the focus for performance analysis and optimisation.

A representative flow chart is shown in Figure 3. Section 1 in the diagram is responsible for partitioning, expanding and re-ordering the source polynomial coefficients into convenient form for large integer multiplication, and is performed once per call. The core processing relates to the diagram sections 2, 3 and 4; and in our implementation it uses an inner and outer loop, executing a total of 8x8 rounds in our Part I implementation, reducing to 4x4 in the Part II implementation. Diagram section 2 is the actual hardware multiply with some surrounding buffer management. Diagram section 3 reduces the resulting polynomial back down to the number of coefficients in an input polynomial, and diagram section 4, modulo adds the interleaved round results to the appropriate coefficients in an overall result accumulator. Finally, diagram section 5 reduces the accumulated result back to the size of a polynomial, then re-orders and reconstructs the actual resulting polynomial.

B. Polynomial Reductions

Multiplying two polynomials as large integers of size N results in an integer of size $2N$; therefore, to fit back into polynomial storage, requires reduction. The form used is a modular reduction by $X^N+ 1$. X^N alone would be an easy discard of the overflow, but the extra '1' in the expression requires the high order (overflow) half of the result to be modularly subtracted from the low (retained) half. Unfortunately, we cannot directly use *multosModularMultiplication*, as in the subtraction, it treats the upper and lower halves of the result as long integers, whereas, Kyber requires a modular subtraction, coefficient-by-coefficient. This necessitated a manual approach to reduction. The *multosModularMultiplication* was still used, but with a modulus that could never be exceeded. Code was added, to provide the modular reduction functionality; creating what was referred to as V1 (unoptimized) code. Note that in

TABLE I: Non-optimized Initial Results (ms)

Type	NTT xform	All xform	Poly vec acc	All Accs	Other Ops	Total Time
NTT						
enc	3	85987	4	41648	6372	134007
dec	2	46863	1	10427	8931	66221
Mult						
enc			4	97975	6610	104585
dec			1	24619	5022	29641

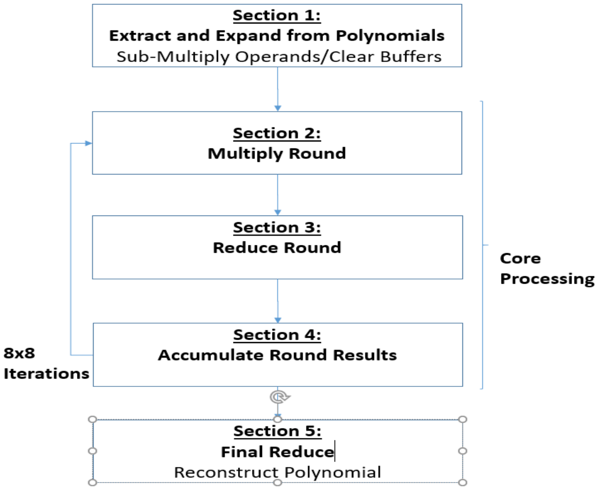

Fig. 3: Flowchart of Polynomial Multiply

Part II of our study, a method was conceived to achieve polynomial reduction via *multosModularMultiplication*, although this relied on assumptions about the inputs, and required changes to personalized data; as discussed in Section VII.

VI. Timing Analysis and Optimisation, Part I

The timing results from the very first (unoptimized) benchmark experiments are shown in Table I.

They include the total execution times for the encryption and decryption processes when computed via the original NTT and alternative multiplier methods. The table also shows the comparative time for the *polyvec_pointwise_acc* and the *polyvec_multiply_acc*. For the NTT case, the time for transforms is shown separately. The NTT encryption method takes an extra 28% of the multiplier method time, but both are far too slow to be usable in almost any practical applications, and so significant optimisation is needed. The *polyvec_pointwise_acc* is significantly faster than the *polyvec_multiply_acc*, but it is reliant on NTT transforms. Both methods spend over 90% of their time in the *polyvec* calculations..

The focus of this research was on the CCoP multiplier method, and the first step was to identify where the main time losses occurred. Experiments were carried out to determine the proportion of total execution time spent in the various processing stages; with the evolving results presented in Table II. The first interesting observation is that in the V1 code, the CCoP multiply section is far from being the bottleneck, accounting for just over 3% of the execution time. Even more interesting is that 76% of the time is taken in manual modular reduction after each multiply round; requiring an inner loop with a modular subtraction/reduction for each coefficient.

TABLE II: polymodmul Performance Breakdown (ms)

Code	Poly Exp	Mult Round	Reduce Round	Acc Round	Poly Reduce	Total
1:V1	412	268	6149	942	360	8131
Part Reduce	437	273	3210	955	329	5204
Block add sub	637	66	2152	936	355	4146
MEL Modulo	419	66	1082	365	370	2302

TABLE III: Modulo Reduction Speed Comparison

Modulo Reduction Technique	Duration (ms)
MULTOS C Remainder	0.97
MULTOS ModularReduction	1.05
MEL Full Tabular	1.08
MEL Partial Tabular	0.21

The V1 code used two modulo reductions per coefficient, per round, relying on the MULTOS C-compiler remainder operator. The modulo reductions served multiple purposes. Firstly, to ensure that the subtraction result was positive and secondly to ensure that the sum of the results from the multiply rounds would not overflow the 32-bit coefficient storage of the accumulator. Using the Kronecker substitution technique, we knew that there would be no carries between result coefficients. Furthermore, as the input coefficient values were 13-bit, the unreduced product of two coefficients could not exceed 26-bits, and with 32 coefficients (in our multiplier round long integers), the maximum number of these that could be added together in a result was 32, equivalent to needing an extra 5-bits. The accumulator could add 8 (3-bits) of these results together, which would require 34-bits total if we were to avoid reduction; thereby overflowing our 32-bit accumulator stores.

As modulo reduction was a necessity within a performance critical part of the code, an experiment was carried out to compare several methods for reducing a 32-bit value mod Kyber_Q. The methods included the *multosModularReduction primitive*, a new tabular technique, described later, along with a partial tabular reduction technique (for reasons which will become apparent). The results are presented in Table III. The remainder operator was the fastest for full reduction; closely followed by *multosModularReduction* and a MEL-optimized tabular version (discussed next);

A. Tabular Reduction

The modulo reduction of result coefficients was sufficiently critical to justify an allocation of the platform's flash memory for tabular reduction methods. The design goals for the latter included, no run-time multiplies, no run-time shifts, data independent run-time and no table needing more than 1kbytes of NVM.

The first attempt used three tables, KTH, KTM and KTL; the first two each have 256, 32-bit values (1kbytes), whereas

TABLE IV: Polymodmul Performance Breakdown (ms)

Version	Poly Mult	Polyvec Mult	indcpa enc	indcpa dec
NTT			134007	66221
First Multiply	8165	24494	104585	29641
Optimized Multiply	2302	6906	35917	12719
Estimated Multiply	1220	3660	22560	9381

KTL has 256, 16-bit (512 bytes). KTH is used for reduction based on the most significant byte of the input, and then KTM and KTL for the next most significant bytes. The contents of the tables are calculated as the maximum integer multiple of KYBER_Q that is smaller than the number represented by the index, assuming the lower unknown bytes are zero. Because the lower bytes could be say all 0xFFs, it is possible that the index value after reduction would still be non-zero; so the tabular reductions for KTH and KTM are used twice in sequence. However, in our implementation we do not need a full reduction. Partial reduction uses KTH once, adds a minimum fixed value to the retained (lower) coefficients and modulo subtracts the upper coefficients, knowing that the results would be positive, and that the addition of multiple multiplication rounds would not overflow the result accumulator. In MULTOS we are abstracted from the underlying processor, so the best code optimisation was to define the functionality in the MULTOS Execution Language (MEL).

B. Other Optimisations

Referring back to diagram Fig 3, the *Accumulate Round* section was the next most time consuming after modular reduction, followed by *Final Reduce*, so efficiencies were sought by unrolling loops, and where possible, using constant rather than run-time calculated addresses. A halt was called on our Part I optimisation due to diminishing returns and the experiments were repeated, also adding the *indcpa_dec* case. The final results are presented in Table IV. The positive aspect is that on our attack-resistant device, the best optimized multiplier code encrypted (decrypted) more than 3.7 (5.2) times faster than the NTT method and 2.9 (2.3) times faster than our initial ported multiplier code. The most negative aspect is that the encryption time is almost 36s, limiting the practical usefulness of the algorithm. To go significantly faster with the Part I implementation design, would require MULTOS to incorporate a larger multiplier (4096 bit) and/or new primitives within its platforms. However, we noted that if a variant of the Kyber768 implementation design could avoid the manual modulo reduction operations (for each multiplication round), we estimated execution times to be 22.56s for encrypt and 9.38s for decrypt. This suggested a worthwhile speed increase, and is what we set out to achieve in Part II of the investigation.

VII. Timing Analysis and Optimisation, Part II

The goal of Part II of the study was to avoid the $X^{\mathrm{N}}+1$ manual reduction of the multiplier round results. The need for this can be seen from Table II, but is illustrated in the power traces of Fig 4. The upper trace in the figure, transitions at the start of each round and the second trace illustrates the stages of a round; the third trace represents the chip power usage. The multiply stage (2) is relatively short, being some buffer preparation and the actual hardware multiply (seen as the peak in power). The accumulate stage (4) takes longer than the multiply and involves adding interleaved results into the correct locations; however, the round reduction stage (3) clearly dominates. The reduction is split between the KYBER_Q reduction of the resulting coefficients (stage 3a) and the subtraction of the overflow coefficients from the retained coefficients (stage 3b). To make a significant improvement to performance, we needed to remove stage (3) entirely, and we began this by revisiting the *multosModularMultiplication* API call in MULTOS.

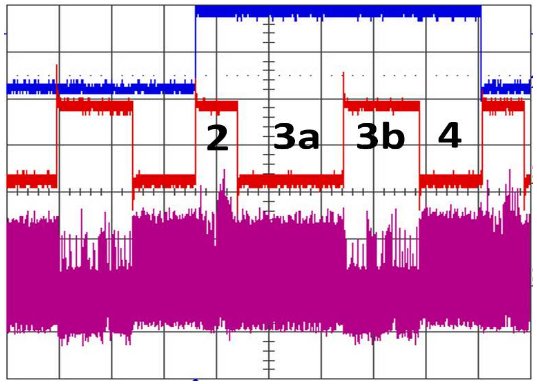

Fig. 4: Stages of Polynomial Multiply (20ms/div)

A. Polynomial Reduction Revisited

Recall from Section V-B that if we use *multosModularMultiplication* to perform a $X^{256}+1$ reduction, it performs an arithmetic subtraction of the overflow large-integer from the retained large-integer, whereas Kyber expects the overflow coefficients to be subtracted (modulo KYBER_Q) coefficient-by-coefficient from the lower retained coefficients. Therefore, to use *multosModularMultiplication* requires that all parts of the overflow integer representing the overflow coefficients, are smaller in arithmetic magnitude than the corresponding values representing the retained coefficients. Adding multiples of KYBER_Q to coefficients will increase their arithmetic representation, without changing the final calculation, however this alone does not help unless we can selectively control the dynamic range of the input data to affect the coefficient results. Therefore the first step was to try and reduce the dynamic range of the data representation for the inputs to the multiplications.

B. Reducing the Representation of Noise

Part I assumed all coefficient values were up to KYBER_Q in size, requiring a minimum of 13-bit storage. When we focussed on encryption, we noted that one of the inputs to the multiplications was always a representation of a noise. Keeping with the original reference implementation, the coefficients in these noise vectors were actually allowed to exceed KYBER_Q, but the extreme values were very limited, being

KYBER_Q +/- 4. The first step in our new method was to vary the way that noise was represented, so that the coefficients could be represented by much smaller arithmetic values. For example, an original noise vector with coefficients in the range 0x1DFD to 0x1E05 would be converted to 0x0 to 0x8 by adding 4 modulo KYBER_Q. If the coefficients in the other input were 13-bit, then multiplying by up to 8 would require 16-bit storage, and if there were up to 32 (5-bit) coefficients in our long integers the largest resulting coefficient would be 21-bits. This gave us the reduction we needed to try and manipulate the overflow coefficient results to be smaller than the retained coefficient results.

C. Noise Data Manipulation: Overflow ¡ Retained Coefficients

The coefficient multiplication results are the sum of products of the input coefficients. To modify the overflow coefficient results differently from the retained coefficients requires a property that sets them apart. The least significant coefficient (say X^0 of the noise) only contributes to the retained coefficients and not the overflow. Therefore, if we could make the X^0 contribution dominant, it would dominate the retained values within the result. To achieve this, requires as a first step, that X^0 is arithmetically much larger than the other noise coefficients; which can be achieved by adding KYBER_Q to it. This alone is not sufficient, as X^0 may be multiplied by a very small coefficient value in the other input. The required second step was to ensure that the other input is stored as its coefficient values plus KYBER_Q. Considering dynamic ranges, one set of coefficient inputs had increased to 14-bits and multiplying this by a noise coefficient (other than X^0) gave a 17-bit result; and 32 of these would take us to 22-bits. The X^0 coefficient multiplication would give us a result in the 26-27bit range; so its contribution dominated and ensured all the retained coefficients were larger than those in the overflow.

D. Input Data Manipulation in Practice

The data conditioning was implemented for encryption (the major performance challenge), and decrypted correctly with the Part I decryption function. It was assumed that noise would be generated in the required form (so no extra processing) and the generator matrix and relevant vectors were available pre-stored in the appropriate form. The technique worked as planned and the first performance results are presented as *64 round noise reduce* within Table VI. Aside from obviating the need for the manual round reduction, the success of the techniques permitted *multosModularMultiplication* to be used for multiplying larger integers, to reduce the number of rounds; as discussed next.

E. Round Reduction

To perform manual round reduction requires that the overflow part of the long-integer multiplication result is accessible, so the inputs can only be half the size of the multiplier capacity. When using *multosModularMultiplication* the hardware and API take care of the reduction so you can use all of the available capacity, which permitted a reduction in the number of rounds in our overall polynomial multiplication. Ideally we would have simply doubled from 32 to 64 coefficients (1028 to 2048 bits), unfortunately, the API required that the inputs were the same size as the modulus, so we needed to store 65 coefficients within 2048bits; a packing/unpacking option was explored as an alternative. From the discussion in Section VII-C we selected 28-bit packing, as sufficient to maintain separation of the result coefficients. It was noted from the outset that the packing and unpacking, would be slow on the secured MULTOS platform, due to the hardware abstraction and defensive coding measures; potentially undermining improvements from the round reduction. However, as packing and unpacking are the kind of utilities that could eventually be incorporated into the MULTOS platform as optimized primitives, it was felt justified to explore further. The best that could be practically achieved in this work was with a MEL-encoded unpack utility; the first results are shown as the *16 round pack/reduce* entry in Table VI. As further round reduction was not possible without a larger multiplier, attention was then turned to the accumulation stage.

F. Round Re-ordering and Accumulations

Accumulation is necessary because of the multi-round approach to multiplication. Because the input coefficients are assigned to long-integers in an interleaved fashion, the accumulator adds results from a round into interleaved places within the accumulated result. Therefore, accumulation is not a simple block addition, if it were, we could use the faster MULTOS primitive *multosADD*. The round calculation was originally formed within an inner and outer loop, indexed by *i* and *j* respectively. It was noted that the locations written to by the accumulator were not unique to a round (16 values), but unique to the sum of *i+j*. Therefore, if the rounds were re-ordered so that common values of *i+j* were sequential, there was the potential of substituting the accumulate with a block addition in some rounds. This assumed that the dynamic range of the stored results could increase without overflow between coefficients. From Section VII-E, we recall that our chosen storage size was 28-bits, yet our maximum round result values were 27-bits, so it was possible to combine two values without overflow. If we had chosen at least 29 bit packing we could have combined four values, which is the most we would need to get optimum benefit from round re-ordering. Table V illustrates the potential for swapping block additions for accumulate operations. The results from the 28-bit version are show in Table VI, as the *16 round plus result ordering entry*. The final entry in the table is a faster estimate of the same functionality, when using a MULTOS primitive unpack utility; based on technical advice from MULTOS.

G. Part I v Part II Results

Table VI summaries the improvements in Part II of the study compared to the original Part I. Note that in Part II, the focus was on polynomial multiplication, and only encryption was considered as a higher level operation; being the maximum challenge for performance.

TABLE V: Swapping Block Adds for Accumulate (ms)

Sum	Normal		28-bit		29-bit	
i+j	Accs	Adds	Accs	Adds	Accs	Adds
0	1	0	1	0	1	0
1	2	0	1	1	1	1
2	3	0	2	1	1	2
3	4	0	2	2	1	3
4	3	0	2	1	1	2
5	2	0	1	1	1	1
6	1	0	1	0	1	0
Totals	16	0	10	6	7	9

TABLE VI: Part I v Part II Results

Rounds: Version	Poly (ms)	Polyvec (ms)	Encrypt (ms)
64:Part I	2302	6906	35917
64:Noise reduce	1149	4130	22318
16:Pack/Reduce	1094	3944	21537
16:Pack/Reduce/Order	996	3631	20420
16:Est. Primitive	935	3282	19692
16:Est. 4096 CCoP	468	1641	9846

VIII. Conclusions and Future Work

Part I of the research confirmed it was possible to load and run the Kyber768 CPAPKE algorithm component on the MULTOS Trust-Anchor, using NTT or 64-round CCoP multiplier solutions; supporting the goal of downloading advanced algorithms for IoT. Although the CCoP solution outperformed NTT, it was too slow for envisaged use. Multiplication of polynomials was identified as the bottleneck, although not due to CCoP multiplications, but the associated application-level reduction and modulo operations. The latter was dominated by manual X^{N} + 1 reduction. Modulo KYBER_Q reductions of individual coefficients were also challenging, as performed many times. Various optimisations were attempted, including a tabular partial reduction, but encryption performance could not improve beyond 36s. The design was changed in Part II by observing that in encryption, one polynomial input is noise-based. The original (reference) noise values varied over a small magnitude range, yet required 13-bit storage; the new proposal offset the noise into the 0 to 8 range, and the least significant coefficient had KYBER_Q added to it. KYBER_Q, was added to all coefficients in the other input, making them non-zero, in the range of 13 to 14 bits. The effect within multiplication was that the least significant coefficient of the noise, dominated the lower coefficients in the result, ensuring they were each larger than the corresponding overflow coefficients. This meant that the modular reduction of *multosModularMultiplication* could be used to generate the same results as in the Kyber NTT reference. In the first instance this removed the manual reduction, and polynomial multiply went from 2302ms to 1149ms. In the second instance it allowed more data to be input to the multiplier, and to decrease the multiplication rounds to 16. This was inefficient at the MULTOS application layer, as it necessitated packing/unpacking before and after multiplication rounds. The fewer rounds, did however make it more practical to order them so that a more efficient *multosADD* could be used instead of the interleaved accumulate function for some rounds. In hindsight, there would have been more benefit with 29 or 30-bit packing, rather than 28. The best practical results, 996ms for a polynomial multiply, and 20420ms for an encryption, cannot be described as fast, however they may be adequate in machine-to-machine scenarios, and with a 4096 bit multiplier, found in similar chips, performance times could halve. The fact that a post-quantum algorithm can be downloaded/run on an existing IoT Trust-Anchor platform, is encouraging for the future security of IoT. It is recommended that the implementation optimisations should be further studied alongside the algorithm design, so the combination ensures the intended strength. It would also be interesting to create a 30-bit packed coefficient solution, and to implement the latest versions of Kyber or similar algorithms, such as Saber [4] (or LightSaber).

Acknowledgments

Enormous thanks are due to Fernando Virdia for expert and patient explanation of post-quantum algorithms, and to Chris Torr of MULTOS for invaluable help and expertise.

References

[1] M. Albrecht, C. Hanser, A. Hoeller, T. Pppelmann, F. Virdia, and A. Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 169–208, 2019.

[2] CC. *Common criteria for information technology security evaluation part1: Introduction and general model, version 3.1 revision 5*, 2017.

[3] D. Chen, N. Mentens, F. Vercauteren, S. Roy, R. Cheung, D. Pao, and I. Verbauwhede. High speed polynomial multiplication architecture for ring LWE and SHE cryptosystems. *IEEE Transactions on Circuits and Systems*, 62(1):157–166, 2015.

[4] J. D'Anvers, A. Karmaker, S. Roy, and F. Vercauteren. Saber: Module LWR based key exchange, CPA secure encryption and CCA secure KEM. In *Africacrypt 2018*, 2018.

[5] GiTHub. *Kyber Reference Code*. https://github.com/pq-crystals/kyber, 2019.

[6] N. Gottert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Proceedings of CHES 2012*, page 512529, 2012.

[7] Infineon. *SLE78CAFX4000PM short product overview, v11.12*. 2012.

[8] L. Kronecker. Grundzge einer arithmetischen theorie der algebraischen grssen. *Journal Fr die reine undangewandteMathematik(92)*, pages 1–122, 1882.

[9] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010*, volume 6110, page 123. Springer, 2010.

[10] K. Mayes, S. Babbage, and A. Maximov. Performance evaluation of the new tuak mobile authentication algorithm, in proc. icons/embedded 2016. pages 38–44, 2016.

[11] K. Mayes and K. Markantonakis, editors. *Smart Cards, Tokens, Security and Applications*, chapter Chapter 17. Springer, 2nd edition, 2017.

[12] MULTOS Organization. *The MULTOS Trust Anchor Development Board*. https://www.multos.com/dev_boards/devboard_details, 2019.

[13] MULTOS Oganization. *website*. http://www.multos.com/, 2020.

[14] NIST Organization. *Submission requirements and evaluation criteria for the Post-Quantum Cryptography standardization process*. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf, 2016.

[15] R. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[16] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. Schanck, G.Seiler, and D. Stehle. Crystals-kyber. *Technical report, National Institute of Standards and Technology*, 2017.

[17] D. Stebila. *Introduction to post-quantum cryptography and learning with errors, summer school on real-world crypto and privacy*. https://www.douglas.stebila.ca/research/presentations, 2018.