

Lattice-based Cryptosystems on FPGA: Parallelization and Comparison using HLS

Timo Zijlstra, Karim Bigou, Arnaud Tisserand

▶ **To cite this version:**

Timo Zijlstra, Karim Bigou, Arnaud Tisserand. Lattice-based Cryptosystems on FPGA: Parallelization and Comparison using HLS. IEEE Transactions on Computers, 2021, 10.1109/TC.2021.3112052 . hal-03347174

HAL Id: hal-03347174
https://hal.science/hal-03347174

Submitted on 17 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Lattice-based Cryptosystems on FPGA: Parallelization and Comparison using HLS

Timo Zijlstra, Karim Bigou and Arnaud Tisserand

***Abstract*—This paper deals with hardware implementations for lattice-based cryptography. Various CPA and CCA secure algorithms for LWE, RLWE and MLWE problems have been studied, parallelized, implemented and compared on FPGA using high-level synthesis. The impact of PRNG choices on the implementations performances and costs is also evaluated. HLS allows us to compare various sets of algorithms, architectures and parameters with a reduced design effort. Our results are often similar to state-of-the-art for various speed and cost trade-offs. Sometimes we obtain better results thanks to the exploration of numerous architecture and algorithm optimizations.**

***Index Terms*—learning with errors, post-quantum cryptography, public-key encryption, hardware implementation, high-level synthesis**

I. Introduction

PUBLIC-KEY cryptography (PKC) uses computationally hard problems to guarantee the security of some cryptographic primitives. The hard problems underlying RSA and ECC can be efficiently solved using *quantum algorithms* [1]. Therefore quantum computers represent a *threat* for services relying on current PKC. Even though it may take decades for a sufficiently large quantum computer to become fully operational, solutions to this security issue should be developed well before that time, to ensure long term security.

Post-quantum cryptography (PQC) is based on mathematical problems for which known quantum algorithms offer no significant speed-up. *Lattice* problems such as *learning with errors* (LWE) [2], *ring*-LWE (RLWE) [3], *module*-LWE (MLWE) [4] and *learning with rounding* (LWR) [5] and its variants are promising examples for PQC.

NIST launched in 2016 a standardization project [6] to select post-quantum algorithms for public-key encryption (PKE) / key-encapsulation mechanism (KEM) and signature. At the second round of this project, 9 schemes are lattice based among the 17 PKE/KEM submissions. At the third round, 3 of the 4 finalists are lattice based for PKE/KEMx}. Such a standardization process requires to estimate the *cost* and *performance* of solutions using real world constraints and different implementation targets. We try to participate to this effort with the implementation on FPGA of various architectures and security levels for several (R/M)LWE based solutions and parameters close to NIST candidates.

T. Zijlstra was with CNRS and Lab-STICC UMR 6285 during this work. He is now with SERMA Safety & Security in Bordeaux, France.

K. Bigou is with University of West Brittany and Lab-STICC UMR 6285, in Brest, France.

A. Tisserand is with CNRS and Lab-STICC UMR 6285 in Lorient, France.

When comparing implementation solutions for PKC, hardware ones usually lead to faster computations and a lower energy consumption than software ones. FPGAs offer a cheap but very effective hardware implementation solution when dealing with wide cost-performance space exploration. Hardware implementations can be realized using *high-level synthesis* (HLS) and input languages such as C for a *reduced design effort*. HLS also allows to quickly explore numerous algorithms, architectures and optimizations solutions which would be tedious with HDL. But, as for HDL implementation, HLS requires some expertise for low-level optimizations especially for finite field arithmetic. HLS was used for instance by [7] for benchmarking authenticated encryption algorithms, and by [8] for the post quantum standardization project.

Hardware implementations are easier to protect against *physical attacks* than software ones. In a previous work [9] (available online), we proposed, implemented and compared on FPGA using HLS several countermeasures against *side channel attacks* (SCAs) for RLWE schemes.

Here, we reuse and extend our polynomial and modular arithmetic units from [9] to:

- design, implement and compare various LWE, RLWE and MLWE hardware solutions with several security parameters for PKE/KEM (inspired from Frodo [10], NewHope [11] and Kyber [12] candidates);
- implement various architectures and study the speed-up achieved for various levels of parallel computations in critical sequences of operations;
- implement the Fujisaki-Okamoto transform [13] to obtain CCA secure KEMs for LWE, RLWE and MLWE;
- evaluate the impact of PRNG and FPGA choices on the performance and cost of the most efficient MLWE PKE with: a fast and small implementation using Trivium [14]; a slower but more secure solution using SHAKE256 [15]; and a hybrid one which uses both PRNGs. In our hybrid solution, Trivium generates the pseudorandom part of the public key while SHAKE256 samples the secret error terms during encryption.

Using the same implementation methodology, tool, target FPGA, design and optimization efforts for all our implementations, we hope to provide a fair comparison between the various evaluated algorithms, parameters and architectures. Our source codes are available as open source [16].

Background on PKE algorithms is recalled in Section III. Sections IV, V and VI respectively detail our LWE, RLWE and MLWE implementations with a focus on parallel computations. Our implementations of the CPA to CCA transformation

and randomness generation methods are described in Section VII. An extensive comparison of our results with state of the art results is provided in Section VIII.

II. Definitions and Notations

- $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$ for $n, q > 0$.
- $\mathcal{R}_q^{k\times m}$ is the $\mathcal{R}_q$-module. An element from this module is $k \times m$ matrix of polynomials in $\mathcal{R}_q$.
- Bold font is used for vectors, matrices and polynomials.
- $\mathcal{B}_\lambda$ denotes the symmetric binomial *distribution* centred around 0 with integer parameter λ.
- $a \xleftarrow{\$} \mathcal{B}_\lambda(\mathbb{Z}_q)$ denotes a random *element* in $\mathbb{Z}_q$ sampled using $\mathcal{B}_\lambda$.
- $\mathbf{a} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$ denotes a random *polynomial* in $\mathcal{R}_q$ whose coefficients are sampled using $\mathcal{B}_\lambda(\mathbb{Z}_q)$.
- $\mathbf{A} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{k\times k})$ denotes a random *matrix* of polynomials sampled using $\mathcal{B}_\lambda(\mathcal{R}_q)$.
- $\odot$ denotes the point-wise multiplication of vectors.
- $\lfloor a \rfloor$ is the integer a' closest to a in $\mathbb{Z}_q$ such that $a' \leq a$.
- $\mathcal{PK}$ and $\mathcal{SK}$ respectively denote the *public* and *secret* keys.
- For e_0 and e_1 two λ-bit integers, we denote $e_0||e_1 := e_0 + 2^{w+w_\lambda} e_1$, where $w = 1+\lfloor \log_2 q \rfloor$ and $w_\lambda = 1+\lfloor \log_2 \lambda \rfloor$.

III. State of the Art

A. Learning with Errors Cryptography

Definition 3.1 (LWE [2]): Let χ be an *error distribution*, that is, a probability distribution close to 0 and symmetric around 0. Let q and m positive integers; $\mathbf{s} \in \mathbb{Z}_q^m$ be some secret vector; and $\mathbf{a}_1, \mathbf{a}_2, \ldots$ some vectors sampled from the uniform distribution over $\mathbb{Z}_q^m$. Then the LWE problem is to find the vector $\mathbf{s}$ given a number of LWE *samples* of the form $(\mathbf{a}_i, \mathbf{a}_i^\intercal \mathbf{s} + e_i)$, where the error terms e_i are sampled from χ.

LWR [5] replaces the random sampling of e_i by a *rounding* mechanism adding a deterministic error to $\mathbf{a}_i^\intercal \mathbf{s}$.

Definition 3.2 (MLWE [4]): For some integer parameter $k > 0$, an MLWE sample for some secret vector of polynomials $\mathbf{s} \in \mathcal{R}_q^k$ is given by some uniformly random vector $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q^k$, together with the polynomial $\mathbf{b} = \mathbf{a}^\intercal \mathbf{s} + \mathbf{e}$, where $\mathbf{e} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$. The search MLWE problem is to find $\mathbf{s}$ given a number of samples.

Note that for $n = 1$, MLWE is similar to LWE with vectors of length k. In MLWE, small matrices and vectors with polynomial coefficients are used. RLWE is obtained by taking $k = 1$. Table I reports the parameters (n, k, m) values used in our LWE, RLWE and MLWE implementations. Works [2], [3], [4] respectively show that LWE, RLWE and MLWE are at least as hard as solving some hard lattice problems using quantum algorithms.

We describe the framework used for instance by NewHope, Kyber and FrodoKEM. Variations of this framework include the use of deterministic errors [17] or Gaussian noise (used in FrodoKEM) instead of sampling the binomial distribution. The secret key $\mathcal{SK}$ is defined by sampling some $\mathbf{s} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{k\times m})$. The corresponding public key $\mathcal{PK}$ is determined by computing a number of LWE/RLWE/MLWE samples for this secret by sampling a uniform random $\mathbf{A} \xleftarrow{\$} \mathcal{R}_q^{k\times k}$ and $\mathbf{e}_0 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{m\times k})$, and computing $\mathcal{PK}$ given by $(\mathbf{A}, \mathbf{b})$ where $\mathbf{b} = \mathbf{s}^\intercal \mathbf{A} + \mathbf{e}_0$. Encryption and decryption algorithms are described in Figures 1 and 2.

Input: Plaintext $\mu \in \{0, \ldots, 2^B\}^{m^2 n}$, $\mathcal{PK} = (\mathbf{A}, \mathbf{b})$
Output: Ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$

$\mathbf{e}_1, \mathbf{e}_2 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{k\times m})$
$\mathbf{e}_3 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{m\times m})$
$\mathbf{c}_1 \leftarrow \mathbf{e}_1^\intercal \mathbf{A} + \mathbf{e}_2^\intercal$
$\mathbf{c}_2 \leftarrow \mathbf{b}\mathbf{e}_1 + \mathbf{e}_3 + \text{ENCODE}_B(\mu)$

Fig. 1. Encryption algorithm $\text{ENC}(\mu, \mathcal{PK})$.

Input: $\mathcal{SK} = \mathbf{s}$, ciphertext $C = (\mathbf{c}_1, \mathbf{c}_2)$
Output: Plaintext μ

$\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1 \mathbf{s}$
$\mu \leftarrow \text{DECODE}_B(\mathbf{d})$

Fig. 2. Decryption algorithm $\text{DEC}(C, \mathcal{SK})$.

The number of bits encoded in each plaintext coefficient is equal to $B + 1$. For RLWE and MLWE, the parameter B is set to zero and $\text{ENCODE}_B(\mu)$ lifts μ to the ring $\mathcal{R}_q$ in a straightforward coefficient-wise manner and returns $\mu \left\lfloor \frac{q}{2} \right\rfloor$. The $\text{DECODE}_B(\mathbf{d})$ function maps coefficients of $\mathbf{d}$ to 0 if they are in the interval $\{\left\lfloor \frac{-q}{4} \right\rfloor, \ldots, \left\lfloor \frac{q}{4} \right\rfloor\}$, else they are mapped to 1. In LWE each coefficient encodes a number of bits $B \geq 1$. Encoding then lifts μ to the module $\mathbb{Z}_q^{m\times m}$ and involves a scalar multiplication by $\left\lfloor \frac{q}{2^{B+1}} \right\rfloor$. Decoding is generalized by dividing $\mathbb{Z}_q$ up into 2^{B+1} intervals as described by [10].

For $n > 1$ and $k = m = 1$, algorithms in Figures 1 and 2 are for the RLWE scheme. Ciphertexts, plaintexts and keys are then polynomials in $\mathcal{R}_q$. For $n = 1$ and $k, m > 1$ the ring $\mathcal{R}_q$ is equal to $\mathbb{Z}_q$ and the plain LWE scheme is obtained, with ciphertexts, plaintext and keys in the form of matrices over $\mathbb{Z}_q$. The intermediate parameter sets for which $n, k > 1$ define the MLWE variant of the scheme.

B. CPA to CCA Conversion

The cryptosystem described in Sec. III-A is secure against *chosen plaintext attacks* (CPA). The Fujisaki-Okamoto transform [13] protects the decryption algorithm against *chosen ciphertext attacks* (CCA). It consists of computing a re-encryption of the decrypted ciphertext and comparing it to the received ciphertext. The decrypted text is returned if and only if the two ciphertexts are equal. If they are not equal, then pseudorandom bits are returned. Since the ciphertext depends on the random error terms generated during encryption, the source of randomness is deterministic and computed by applying a hash function to the message and the public key. A third argument is added to the ENC function that specifies the source of randomness used for the binomial sampling. The CCA-secure algorithms in Figures 3 and 4 use hash function H and G, and Algorithms 1 and 2. This CPA to CCA transformation and variants of it are used by Kyber, NewHope, Frodo and other lattice-based key exchange mechanisms.

Input: $\mathcal{PK}$, random $\mu \in \{0,1\}^n$
Output: Ciphertext C and session key K

1: $(r_1, r_2) \leftarrow H(\mathcal{PK}||\mu)$
2: $C \leftarrow \text{ENC}(\mu, \mathcal{PK}, r_1)$
3: $K \leftarrow G(C||r_2)$

Fig. 3. CCA-secure encapsulation function ENCAPS.

Input: Ciphertext C, $\mathcal{PK}$ and $\mathcal{SK}$
Output: Session key K'

1: $\mu' \leftarrow \text{DEC}(C, \mathcal{SK})$
2: $(r'_1, r'_2) \leftarrow H(\mathcal{PK}||\mu')$
3: $C' \leftarrow \text{ENC}(\mu', \mathcal{PK}, r'_1)$
4: **if** $C' = C$ **then**
5: $\quad K' \leftarrow G(C'||r'_2)$
6: **else**
7: $\quad K' \xleftarrow{\$} \{0,1\}^{256}$
8: **end if**

Fig. 4. CCA-secure decapsulation function DECAPS.

C. Implementation of Main Operations

1) Matrix multiplication: The multiplication of the public key $\mathbf{A}$ with the error matrix $\mathbf{e}_1$ is the most expensive operation in the standard LWE scheme. It consists of k^2m multiplications in the rings $\mathbb{Z}_{2^{15}}$ or $\mathbb{Z}_{2^{16}}$. In [18] the matrix multiplication is accelerated by computing partial products in parallel using up to 16 DSP blocks.

2) Polynomial multiplication: In RLWE and MLWE, the most expensive arithmetic operation is the polynomial multiplication. Multiplication in the ring $\mathcal{R}_q$ is computed using the *number theoretic transform* (NTT). FPGA implementations of NewHope using the NTT for $n = 1024$ are given by [19] and [20]. A fast and area optimized implementation for $n = 256$ is given by [21]. In [22] the usage of HLS for the implementation of the NTT is discussed. While implementations using schoolbook polynomial multiplication have been proposed in [23], [24], they are much slower than the NTT.

In [24] the schoolbook algorithm is optimized for coefficient multiplication on Xilinx DSP48E blocks with 18×25-bit hardwired integer multiplication. The coefficients sampled from the error distribution can be represented on a few bits. Therefore, a naive multiplication of a coefficient with $w = \lfloor \log q \rfloor + 1$ bits by an error coefficient would underuse the DSP block. Paper [24] "packs" two error coefficients e_0, e_1 of size $w_\lambda = 1 + \lfloor \log_2 \lambda \rfloor$ into a new $(w + 2w_\lambda)$-bit coefficient $e_0 + 2^{w+w_\lambda} e_1$. If $(w + 2w_\lambda) < 25$, then for any w-bit coefficient a the multiplication $(e_0 + 2^{w+w_\lambda} e_1)a$ can be computed on one DSP block. The product $e_0 a$ can be read on the first $w + w_\lambda$ LSBs of the output. The product $e_1 a$ is obtained by applying $w + w_\lambda$ right shifts to the output and again selecting the $w + w_\lambda$ LSBs of the remainder. The sign of the products is computed separately. Then two multiplications are obtained for the cost of one.

3) Binomial sampling: The $\mathcal{B}_\lambda(\mathbb{Z}_q)$ distribution is sampled by generating 2λ random bits $x_1, \ldots, x_\lambda, y_1, \ldots, y_\lambda$ and computing $\sum_{i=1}^{\lambda} x_i - y_i \bmod q$. The sampling requires 2λ random bits per coefficient. For a total of $mn(2k+m)$ coefficients for the 3 errors $\mathbf{e}_1, \mathbf{e}_2$ and $\mathbf{e}_3$, the amount of random bits needed is considerable. In the specifications of most of the NIST round 2 candidates it is suggested to use SHAKE256 or AES to supply the randomness. Some implementations however, such as [18], use Trivium because it is faster. Precomputing random bits and storing them in BRAM is used in [25] to improve the throughput of the PRNG.

4) Modular reduction: In the LWE scheme the modulus is a power of 2, so that no computation is required to compute modular reduction. In RLWE/MLWE however, for fast polynomial arithmetic, one often chooses to use the NTT which requires the existence of a $2n$-th root of unity in $\mathbb{Z}_q$. This is the case if q is a prime for which $q \equiv 1 \bmod 2n$. The choices for q are therefore limited. For prime moduli of the form $q = 2^{l_1} - 2^{l_2} + 1$ for some integers l_1 and l_2, one has $2^{l_1} - 2^{l_2} + 1 \equiv 1 \bmod 2n$ if $l_2 \geq \log_2(2n)$. For $n = 256$ suitable primes include $7681 = 2^{13} - 2^9 + 1$ which is used in the original version of Kyber and for $n = 1024$ the prime $q = 2^{14} - 2^{12} + 1 = 12289$ is used in NewHope. We use a modular reduction method in the style of [26] for moduli of the form $2^{l_1} - 2^{l_2} + 1$. Using the fact that $2^{l_1} \equiv 2^{l_2} - 1 \bmod q$, a modular reduction can be computed using only bitwise shifts, additions and subtractions.

D. Parameters Selected for our Implementations

We implement the CPA and CCA secure LWE, RLWE and MLWE schemes for the parameter sets shown in Table I.

We choose LWE parameters from FrodoKEM [10] except for the Gaussian distribution. We sample the $\mathcal{B}_\lambda$ distribution instead, where λ is chosen such that the obtained $\mathcal{B}_\lambda$ distributions are close to the Gaussian distributions from FrodoKEM. This allows us to make a fair comparison between LWE on one hand and RLWE and MLWE (both using binomial distributions) on the other. To the best of our knowledge, there does not exist any attack that exploits the small difference between the sampled distribution and the Gaussian distribution used in the security proof. The performance of the best algorithms solving LWE does not depend on the exact error distribution, which is why schemes such as Kyber [12] also prefer binomial sampling.

Our parameters for RLWE and MLWE are those used by NewHope [11] and Kyber [12] respectively. A newer version of Kyber [27] proposes to use the modulus $q = 3329$. On FPGA, there is hardly any speedup from replacing 13-bit operands by 12-bit when this does not reduce the number of required DSP blocks. Reducing q however, requires to implement quadratic extension field arithmetic. To avoid the overhead in computation time that this would cause, we first implement the original scheme using $q = 7681$. In section VI, we also implement the scheme with the new modulus and compared its performance to the original scheme.

The parameter sets are designed for the NIST security levels 1, 3 and 5, where level 1 corresponds to AES-128, 3 to AES-192 and 5 to AES-256. Level 1 is claimed by [10] for Frodo using parameter set LWE-640, and by [12] for Kyber using parameters set MLWE-512. Level 3 proposals use parameter

TABLE I
PARAMETER SETS USED IN OUR IMPLEMENTATIONS.

Scheme	n	m	k	q	λ
LWE	1	8	640 / 976 / 1344	2^{15} / 2^{16} / 2^{16}	15 / 10 / 4
RLWE	1024	1	1	12289	8
MLWE	256	1	2 / 3 / 4	7681	5 / 4 / 3

sets LWE-976 and MLWE-768, while LWE-1344, RLWE-1024 and MLWE-1024 are used in level 5.

IV. FPGA IMPLEMENTATION OF LWE

In Sections IV to VIII, we use Vivado HLS (version 2018.1) on an Artix-7 FPGA (XC7A200) from Xilinx for all our implementations since this FPGA family is frequently used in other works (see Table VII in Section VIII). We also verified several of our architectures for CPA-secure RLWE on a ZedBoard card with a Zynq XC7Z020 FPGA (we do not have an Artix-7 card) where performance and cost results obtained on the Zynq FPGA card accurately confirm the synthesis and place&route (SPR) results. Below we report SPR results on the Artix-7 family.

A. Matrix Arithmetic for LWE

We extend the method from [24] to speed-up schoolbook polynomial multiplication, described in Sec. III-C, to the matrix multiplication for the standard LWE scheme. Matrices $\mathbf{A}$ and $\mathbf{e}_1$ coefficients are 15 and $w_\lambda = 1 + \lfloor \log_2 \lambda \rfloor$ bits wide respectively. We pack two coefficients $e_{00}||e_{10}$ to reduce the $8 \times k$ matrix $\mathbf{e}_1$ with w_λ-bit elements to a $4 \times k$ matrix with $(w + 2w_\lambda)$-bit ones. Then multiplying one coefficient from $\mathbf{A}$ by one from $\mathbf{e}_1$ requires a single DSP block.

The coefficients of the public key matrix $\mathbf{A}$ are generated by the PRNG. At each clock cycle, one coefficient is generated. During the first clock cycle, a_{00} is generated and multiplied by all 4 coefficients in the first column vector of $\mathbf{e}_1$.

The resulting vector is added to the first column of the output matrix. All the coefficients that are loaded in the first clock cycle are coloured blue in Figure 5. During the second clock cycle, the red coefficients are loaded. The resulting integer products are all added to the first column of the output

$$\underbrace{\begin{pmatrix} e_{00}||e_{10} & e_{01}||e_{11} & \cdots \\ \vdots & \vdots & \\ e_{60}||e_{70} & e_{61}||e_{71} & \end{pmatrix}}_{k \text{ columns}} \times \underbrace{\begin{pmatrix} a_{00} & a_{01} & \cdots \\ a_{10} & a_{11} & \\ \vdots & & \\ \vdots & & \\ a_{(k-1)0} & & \end{pmatrix}}_{k \text{ columns}}$$

$$= \underbrace{\begin{pmatrix} c_{00} & c_{01} & \cdots \\ c_{10} & & \\ \vdots & \vdots & \\ c_{70} & & \end{pmatrix}}_{k \text{ columns}}$$

Fig. 5. Matrix multiplication $\mathbf{e}_1^\intercal \mathbf{A}$: each element of $\mathbf{A}$ is multiplied with a column vector of $\mathbf{e}_1^\intercal$.

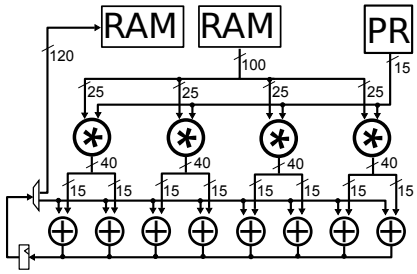


Fig. 6. Architecture for matrix multiplication $\mathbf{e}_1^\intercal \mathbf{A}$. The 4 DSPs compute 8 integer products. Coefficients of $\mathbf{A}$ are generated by the PRNG (denoted PR).

matrix. The first column of this output matrix is completely computed after k (+ pipeline depth) clock cycles. Only then the computation of the second column begins.

Each row of $\mathbf{e}_1$ is stored in one BRAM (i.e. matrix $\mathbf{e}_1$ uses 4 BRAMs). The architecture of the matrix multiplication is shown in Figure 6. To increase the level of parallelism by a factor two, the blue and red multiplications can be performed at the same time. Then twice as many DSP blocks are required for the matrix multiplication and two coefficients of $\mathbf{A}$ have to be generated at the same time. For higher degrees of parallelism, multiple elements on the same row of $\mathbf{e}_1$ have to be read simultaneously. Therefore the rows of $\mathbf{e}_1$ have to be implemented on multiple BRAMs each.

B. Parallelization using HLS

The C source code of the matrix multiplication $\mathbf{c}_1 \leftarrow \mathbf{e}_1^\intercal \mathbf{A}$, illustrated in Figure 5, is reported in Figure 7. The loops labelled `col_A` and `row_A` iterate over the columns and rows of A respectively. Column of the output matrix are loaded and stored by loops `copy1` and `copy2`. The `prng` function generates the next coefficient of A, and `comp_2prods` computes $a \cdot (e||e')$ for coefficients a, e, e' using the error encoding method described in paragraph III-C2.

In order to specify optimization to the HLS tool, we use various *directives* (see [28]). Applying the `pipeline` directive to the loop `row_A`, ensures that this loop is pipelined and the subloop `row_E` is completely unrolled. That is, all 4 iterations of the loop `row_E` are computed at the same time on 4 DSPs. Arrays are implemented on a single BRAM by default. Without any specifications, the HLS tool would try to implement `E1` on a single BRAM. However, all 4 elements of each column have to be loaded simultaneously. Therefore we use the `array_partition` directive on `E1` to partition the local memory into 4 parallel BRAMs.

We parallelize the computation even further by applying the `unroll` directive on loop `row_A` using several unrolling factors: 2, 4, 8 and 16. For unrolling factors 4, 8 and 16, multiple elements on the same row have to be accessed at the same time. Therefore the array `E1` has to be partitioned in the second dimension as well, using the `array_partition` directive, to prevent simultaneous accesses to the same BRAM. A similar effect can be obtained using the `array_reshape` directive, which results in fewer additional BRAMs than `array_partition`. Using the `array_map` directive, multiple arrays can be implemented on one single BRAM. This

TABLE II
CPA-SECURE LWE IMPLEMENTATIONS RESULTS FOR ENCRYPTION AND DECRYPTION.

Scheme LWE-k	Freq. MHz	Time μs enc / dec	Area DSP, BRAM, Slices, LUT
LWE-640	200	2275 / 232	6, 16, 1629, 4311
LWE-976	200	5123 / 353	8, 16, 1601, 4322
LWE-1344	200	9506 / 486	6, 25, 1439, 3832

```
col_A: for(i=0; i<k; i++){
    copy1: for(ii=0; ii<8; ii++)
       C1_tmp[ii] = C1[ii][i];  // copy BRAM -> registers
    row_A: for(jj=0; jj<k; jj++){
        sum = 0;
        prng(State_A, &a_coeff);    // PK coeff. from PRNG
        row_E: for(j=0; j<4; j++){
            comp_2prods(a_coeff, E1[j][jj], &prod1, &prod2);
            C1_tmp[2*j] = C1_tmp[2*j] + prod1; // update C1
            C1_tmp[2*j+1] = C1_tmp[2*j+1] + prod2;
        }
    }
    copy2 :for(ii=0; ii<8; ii++)
        C1[ii][i] = C1_tmp[ii];  // copy registers -> BRAM
}
```

Fig. 7. Source code for matrix multiplication $\mathbf{c}_1 \leftarrow \mathbf{e}_1^\intercal \mathbf{A}$.

may have a negative impact on the computation time, as some arrays need to be accessed during the same clock cycle.

Other examples of directives used in our implementations include`inline`, `allocation` and `dependence`. To find the optimal choice of directives, we have tried dozens of directive combinations including

- different factors for `array_partition`, `unroll` and `array_reshape` with different options,
- pipelining using different initiation intervals,
- `array_map` for different variables,
- `inline` and `dependence` for various loops and functions,
- `allocation` to set a strict limit to the number of DSPs used by the implementation for various limits.

The same (somewhat) exhaustive approach to the exploration of the design space is used for RLWE and MLWE implementations in sections V and VI. In total we have tested over 100 different configurations, which would not be possible in reasonable time using HDL (and debugging so many architectures would be tedious). Clearly, this demonstrates the interest of HLS for PKC implementations. The complete set of directives used for our source codes is available online in our repository [16].

C. Implementation results

The implementation results are reported in Table II. The error encoding technique for packing two error terms in one $w + 2w_\lambda$ bits integer allows to compute 8 multiplications in parallel using 4 DSP blocks for the parameters sets of $k = 640$ and $k = 1344$. For $k = 976$ however, the error terms are still 5-bit integers while the coefficient size is increased to 16 bits (see Table I). Therefore $w + w_\lambda > 25$ and extra DSP blocks

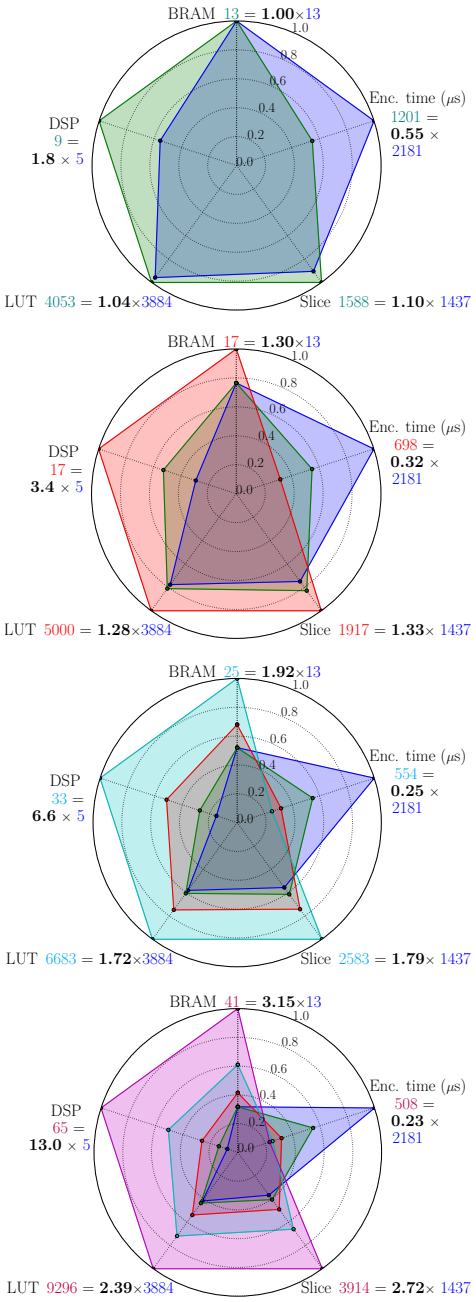


Fig. 8. Comparison of the LWE-640 base implementation (blue) to parallel ones with unrolling factors 2 (green), 4 (red), 8 (cyan) and 16 (magenta).

are needed for the multiplications. For $k = 1344$ the size of the error terms decreases to 4 bits.

The matrix multiplication using 4 parallel DSP blocks is computed in roughly $k^2 = 409600$ cycles for $k = 640$. This operation takes up 90 percent of the total encryption time. The impact of parallelism on the timing and area implementation results is shown in Figure 8. These results are for the LWE-640 encryption algorithm only. The unrolling factor 2 divides the total encryption time by almost 2 while the overhead in terms of DSPs is lower than 2. In terms of slices, LUTs and BRAMs, the trade-off is even more favourable for the parallelized implementation, which even holds for the more parallelized implementations using unrolling factors 4, 8 and 16. For these

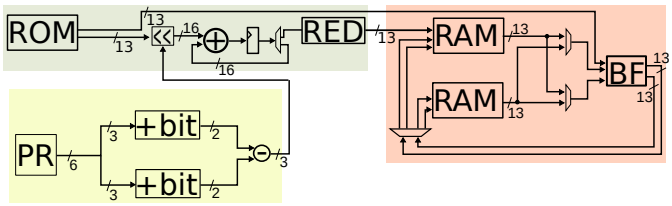


Fig. 9. Binomial sampler (yellow part) using PRNG for the error polynomials in the NTT domain (red part). The negative wrapped convolution is computed using a shift and add based multiplier (green part), exploiting the fact that the binomial samples are small. The NTT uses one butterfly operator (BF) that consists of a single DSP block and integer addition/subtraction operators.

TABLE III
CPA-SECURE RLWE-1024 ENCRYPTION/DECRYPTION RESULTS.

Version	Freq. MHz	Time μs enc / dec	Area DSP, BRAM, Slices, LUT
Our work in [9]	250	65 / 39	7, 12, 4106, 11164
Sequential	206	110 / 47	1, 11, 3820, 10563
Parallel NTTs	258	63 / 38	4, 10, 3701, 10112
Unrolled	251	59 / 35	6, 16, 4474, 12301

higher degrees of parallelism however, the number of DSPs increases faster than the computation time decreases. For an unrolling factor of 8 and 16 the obtained frequency is reduced, limiting the obtained speed-up. More detailed results including comparisons with results from the state of the art are reported in Table VI in Section VIII.

V. NEW OPTIMIZED RLWE IMPLEMENTATIONS

We re-used the finite field and polynomial arithmetic units from our previous work [9], and we added the implementation of the CPA secure RLWE cryptoscheme for $n = 256$ and $n = 1024$. Our new architectures include modular reduction, NTT and the binomial sampler, as illustrated in figure 9. The negative wrapped convolution computes the products of small error terms with the $2n$-th roots of unity. These multiplications are computed using a shift and add based multiplier. The constant geometry variant [29] of the NTT is used. Bit-reversal is avoided by using the decimation-in-frequency (DIF) algorithm for the forward NTT and decimation-in-time (DIT) for the inverse, as proposed in [30].

Our new (CPA and area optimized) implementation results reported in Table III are compared to our non-CPA version from [9] and RLWE-1024. The sequential architecture requires only one DSP block but doubles the encryption time. In the parallel architecture, 2 forward NTTs are computed simultaneously with $\frac{n}{2}\log n$ less cycles in the encryption function, resulting in a smaller latency. The parallel architecture also leads to a higher frequency than the sequential one, resulting in an even more significant speed-up. The unrolled architecture unrolls loops of the point-wise computations by a factor 2. There is a small speedup compared to the parallel architecture but with some area overhead in terms of DSPs and BRAMs. Thanks to HLS, adding CPA protection was possible in a moderate design time.

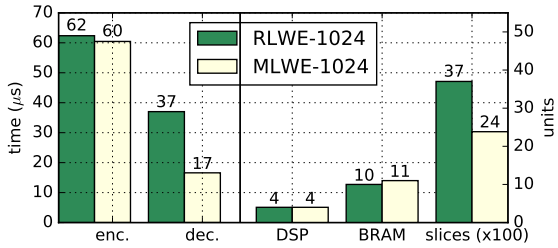


Fig. 10. Comparison of CPA-secure MLWE (using $q = 7681$) with RLWE ($q = 12289$) PKE FPGA implementations for the same security level.

VI. MLWE IMPLEMENTATIONS AND COMPARISON

A. Modifying the RLWE Implementation for MLWE

We transform our RLWE-1024 implementation for MLWE using slight changes, starting by changing n from 1024 to 256. The arithmetic units are re-used for computations in $\mathcal{R}_q$ and MLWE. This includes our architecture in Figure 9 (modified for $n = 256$) that generates binomial samples in the NTT domain and now denoted BN. The same operations are performed but on polynomial coefficients of k-dimensional vectors over $\mathcal{R}_q$. The MLWE scheme is thus implemented by applying the operators used in RLWE to each of the k polynomials (each of degree n) of the vectors in a sequential manner. This is achieved by modifying the control accordingly. Each vector consists of $14 \cdot 256 \cdot k$ bits and is stored in one 18 kb BRAM. For $k = 4$, around 14 kb are used in each BRAM, while for $k = 2$, only 7 kb are used. In a sequential architecture, the number of BRAMs is the same for $k \in \{2, 3, 4\}$. Extra additions and a modified control are needed to support the multiplication of matrices and vectors of dimension k. To avoid storing the $k \times k$ random matrix $\mathbf{A}$, which is part of the public key, we use the PRNG to generate the coefficients of the polynomials in matrix $\mathbf{A}$ on the fly, as suggested by [12]. The public key to be stored in the architecture only consists of the vector $\mathbf{b} \in \mathcal{R}_q^k$ and the seed for the PRNG. We apply one step of rejection sampling in order to avoid too much bias in the distribution of the coefficients (see section VII-A), as proposed for instance for Kyber in [12].

As seen in Sec. III-D, several values of the security parameter $k \in \{2, 3, 4\}$ are used for MLWE based candidates. We provide implementations for all those values with results in Table IV. Having a generic source code for HLS allows us to select k easily. In HDL, this would not be simple to optimize the performances using pipelining. k determines the number of required multiplications in $\mathcal{R}_q$. During the encryption, $k^2 + k$ multiplications in $\mathcal{R}_q$ and $2k$ forward NTTs are needed. The decryption consists of k multiplications in $\mathcal{R}_q$, with only one inverse NTT.

Figure 10 compares, for a similar security level, RLWE with $n = 1024$ and MLWE with $k = 4$ implementations. For encryption, MLWE is slightly faster but for decryption MLWE is twice as fast as RLWE with only one additional BRAM and even less slices. The impact of k on the decryption time of MLWE is limited, since only the size of the computation $\mathbf{c}_1 \cdot \mathbf{s}$ depends on k. During the encryption however $k^2 + k$

TABLE IV
CPA-SECURE MLWE FPGA IMPLEMENTATIONS FOR DIFFERENT SECURITY LEVELS.

Scheme $(k \times n)$	Freq. MHz	Time μs enc / dec	Area DSP, BRAM, Slices, LUT
$q = 7681$			
MLWE-512	256	30 / 12	4, 11, 2380, 5538
MLWE-768	256	44 / 15	4, 11, 2540, 6031
MLWE-1024	250	61 / 17	4, 11, 2383, 5515
$q = 3329$			
MLWE-512	227	61 / 19	13, 19, 6696, 17291
MLWE-768	227	98 / 23	13, 19, 7036, 17288
MLWE-1024	222	144 / 29	13, 19, 6975, 17187

multiplications in $\mathcal{R}_q$ and $2k$ NTTs have to be computed. The encryption time is therefore heavily impacted by increasing the parameter k. In order to estimate the performance of Kyber from the Round 2 specification, we modify the implementation by using modulus $q = 3329$. The fast modular reduction algorithm for modulus 7681 cannot be used and must be replaced. Moreover, there are no $2n$-roots of unity in $\mathbb{Z}_{3329}$, therefore the NTT has to be modified as well, and quadratic extension field arithmetic has to be added to replace the point-wise multiplications. As a result, the latency is increased and extra DSP blocks are required as reported in Table IV. Adding a quadratic extension for the field arithmetic is not simple. But using HLS this corresponds to a few days of design and debug (using HDL this would be tedious).

B. Parallelization of Operations in $\mathcal{R}_q^k$

We also propose parallelized implementations of MLWE encryption and decryption whose computation time is independent of the vector length k. While the computations in $\mathbb{Z}_q$ are still performed sequentially, hardware is added to compute the operations on a higher level (matrix-vector operations) in parallel. During the encryption the k components of the error vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ have to be sent to the NTT domain. All of these $2k$ transforms are computed simultaneously. The operation (for $k = 2$)

$$\mathbf{e}_1, \mathbf{e}_2 \mapsto \begin{pmatrix} \mathrm{NTT}(\mathbf{e}_1^{(0)}) \\ \mathrm{NTT}(\mathbf{e}_1^{(1)}) \end{pmatrix}, \begin{pmatrix} \mathrm{NTT}(\mathbf{e}_2^{(0)}) \\ \mathrm{NTT}(\mathbf{e}_2^{(1)}) \end{pmatrix}$$

is computed in the time it takes to compute one NTT, that is, $\frac{n}{2}\log(n)+\delta$ cycles for where δ is the pipeline depth. Similarly, PRNGs and binomial samplers are added to sample the $2k$ error polynomials simultaneously. The k^2 multiplications in $\mathcal{R}_q$ for the computation of $\mathbf{c}_1 \leftarrow \mathbf{A}^\intercal\mathbf{e}_1 + \mathbf{e}_2$ and the k multiplications in $\mathcal{R}_q$ needed to compute $\mathbf{c}_2$ are also computed in parallel. For $k = 2$, the operation

$$\mathbf{e}_1, \mathbf{A} \mapsto \begin{pmatrix} \mathbf{a}^{(00)} \odot \mathbf{e}_1^{(0)} + \mathbf{a}^{(01)} \odot \mathbf{e}_1^{(1)} \\ \mathbf{a}^{(10)} \odot \mathbf{e}_1^{(0)} + \mathbf{a}^{(11)} \odot \mathbf{e}_1^{(1)} \end{pmatrix}$$

is computed in just over n cycles, which is the time it takes to compute one single point-wise multiplication. For the computation of $\mathbf{A}^\intercal\mathbf{e}_1$, in order to compute the k^2 multiplications over $\mathcal{R}_q$ in parallel, we need to access all k^2 coefficients of $\mathbf{A}$ at the same time. Therefore, we generate a seed for the PRNG for each of the k^2 coefficients of $\mathbf{A}$. The public key then consists of a vector $\mathbf{b} \in \mathcal{R}_q^k$ and a seed for the PRNG used to generate the k^2 seeds for the $k \times k$ matrix $\mathbf{A}$. The parallel architecture for $k = 3$ using the BN unit described in the previous section is shown in Figure 11.

C. Parallel Implementation using HLS

The C source code in Figure 12 is an excerpt of the MLWE encryption. It computes the matrix-vector product $\mathbf{A}^\intercal\mathbf{e}_1$ where all the matrix and vector coefficients are in the NTT domain.

A standard matrix-vector product can be recognized in the loops labelled `col` and `row`. The `coeff` loop iterates over the coefficients of the polynomials in matrices $\mathbf{A}$ and $\mathbf{e}_1$. The matrix $\mathbf{A}$ is not read from memory, but computed "on the fly". The k^2 internal PRNG states are read from memory and the PRNG is used to generate the coefficients of the k^2 polynomials in $\mathbf{A}$. The `reduce` and `reduce_fast` functions perform modular reduction, the `prng` function samples a 13-bit signed integer, and the `DW` macro casts the operands of the multiplication to the `int26` type, to get a 26-bit signed integer as result.

In order to generate a parallel architecture, some directives have to be specified accordingly in Vivado HLS. To compute all of the k^2 polynomial multiplications simultaneously, we set the `pipeline` directive on the `coeff` loop. It forces all subloops to be completely unrolled. Then the k^2 operations in the `col` and `row` loops are performed in parallel.

We use the `array_partition` directive to partition the arrays `E1[k][n]` and `C1[k][n]` into k different arrays. This distributes them over k different BRAMs each. Then k values can be loaded from the array `E1` at the same time and k values can be written to `C1` at the same time. We apply the same directive to both dimensions of the $k \times k$ array `Trivium_States`. Clearly doing this type of architecture exploration in HDL would be labor intensive.

To generate an architecture for a different vector length k, we use a SageMath script that creates a new header file defining k and computes a new set of valid keys (the C preprocessor in Vivado HLS is not able to perform such mathematical computations). The C source code remains the same and the same directives apply. Our SageMath script also generates all the constants used in the architecture, such as n-th roots of unity and exponents parametrizing the modulus. A simple change of parameters in the script is all that is needed to generate architectures for different values of (n, q, k) without changing the C source code. We can even switch between RLWE ($k = 1$) and MLWE implementations ($k = 2, 3$ or 4) by simply generating a new header file. For area optimization, we add some specific directives depending on the parameter k. The `allocation` directive for instance, allows to set a limit to the number of DSP blocks in the implementation.

D. Implementation Results

The PRNG is instantiated with the Trivium stream cipher [14]. The results are shown in Table V. In the parallel MLWE implementation, the impact of k on the encryption time is mitigated by adding BRAMs and DSP blocks. The latency

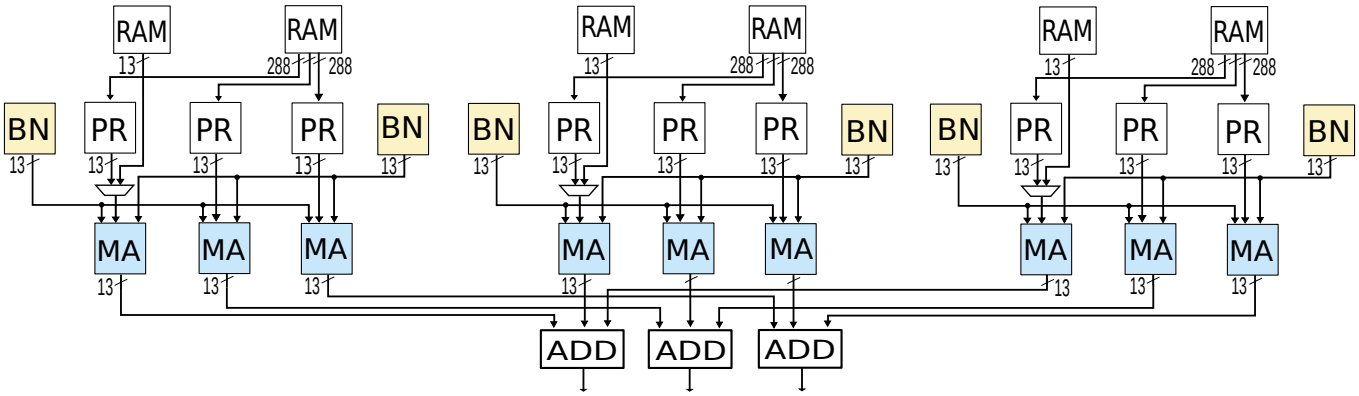


Fig. 11. Proposed parallel architecture for the matrix-vector multiplication in MLWE-768. The PRNG ("PR") generating matrix $\mathbf{A}$ uses the internal PRNG states stored in RAM. The modular arithmetic unit ("MA") computes modular multiplication and addition with the error coefficients supplied by the BN units. The polynomial products are summed up to obtain $\mathbf{c}_1 = \mathbf{A}^\intercal \mathbf{e}_1 + \mathbf{e}_2$.

```
coeff: for(i=0; i<N; i++){
    col: for(jj=0; jj<K; jj++){
        c1_coeff = 0;
        row: for(j=0; j<K; j++){
            A_coeff = 0;
            prng(Trivium_States[j][jj], &A_coeff);
            c1_coeff += reduce(DW(A_coeff)*DW(E1[j][i]));
        }
        C1[jj][i] = reduce_fast(c1_coeff);
    }
}
```

Fig. 12. HLS code for matrix-vector multiplication $\mathbf{A}^\intercal \mathbf{e}_1$.

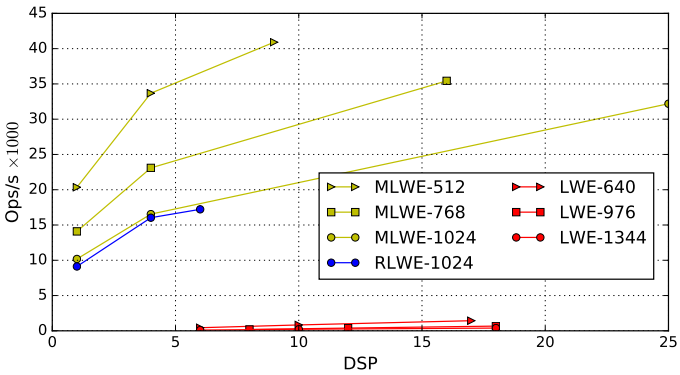


Fig. 13. Throughput (in k-encryptions per second) *vs* area (in DSPs) trade-offs for various parallelism levels. The most left point of each curve corresponds to a sequential architecture, the middle point embeds parallel NTTs (for RLWE/MLWE) and the most right point is a full parallel architecture.

(in clock cycles) of the arithmetic part of the scheme is then the same for $k = 2, 3$ and 4. A slight increase in encryption and decryption time is due to the loading and storing of public keys and ciphertexts of increased size. In Table V, increasing k means adding n cycles to the decryption latency, during which the $k \cdot n$ coefficients of the ciphertext part $\mathbf{c}_1$ are loaded. The encryption latency increases by $2n$ cycles since both $\mathbf{b}$ and $\mathbf{c}_1$ consist of $k \cdot n$ coefficients.

The throughput and area (in DSP blocks) trade-offs of our LWE, RLWE and MLWE implementations, with various parallelism levels, are shown in Figure 13. Sequential architectures for RLWE and MLWE using only 1 DSP block and no other optimizations than pipelining are compared to the slightly parallel (computing NTTs simultaneously) and full parallel ones. For RLWE and MLWE implementations, the throughput is increased by computing parallel NTTs during the encryption. Further parallelism is obtained by unrolling loops. This almost doubles the throughput of the slightly parallel architecture for MLWE-1024. However for RLWE-1024, the speedup is limited to only 7 percent compared to the slightly parallel version. This is due to the memory access patterns of the NTTs which limit further parallelism. In RLWE these NTTs consist of 10 stages of 512 butterfly operations each, while in MLWE only 8×128 butterfly operations are necessary. The potential for parallel architectures provided by the matrix structure, is clearly an advantage for MLWE compared to RLWE. For the LWE implementations, the throughput increases when unrolling the matrix multiplication loop. It remains however, far below those of MLWE and RLWE.

Obtaining optimized implementations for such a set of architectures and algorithms in HDL would have been very difficult and costly.

TABLE V
PARALLEL CPA-SECURE MLWE ($q = 7681$) ARCHITECTURE RESULTS FOR DIFFERENT SECURITY LEVELS.

Size k	Freq. MHz	Time μs enc / dec	Area DSP, BRAM, Slices, LUT
2	204	25 / 14	9, 17, 4565, 8584
3	196	29 / 16	16, 25, 6271, 12383
4	196	32 / 17	25, 29, 8988, 16803

VII. Randomness Generation and CCA Implementations

In this section, we investigate various links between area, performances and security of our solutions. We evaluate the use of: rejection sampling for public-key generation; a more secure PRNG; CCA transformation for higher security.

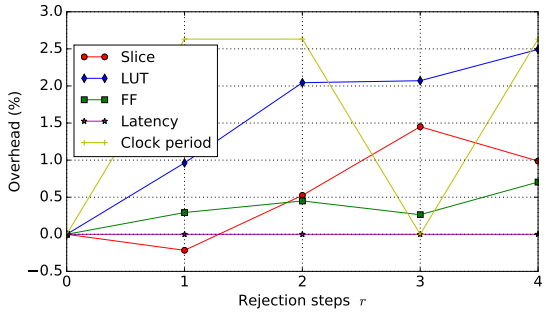


Fig. 14. Area/timing overhead of CPA-secure MLWE-1024 implementation due to rejection sampling.

A. Rejection Sampling

To generate the coefficients of the public key $\mathbf{A}$, uniform sampling over $\mathbb{Z}_q$ is needed. The naive way for sampling the uniform distribution over $\mathbb{Z}_q$ is to generate $w = \lceil \log_2(q) \rceil$ random bits defining a w-bit number a and returning $a \bmod q$. This results in a biased distribution: for any $a_0 \in \{0, \ldots, 2^w - q - 1\}$ and $a_1 \in \{2^w - q, \ldots, q - 1\}$, the probability of obtaining a_0 is twice as high as the probability of obtaining a_1. The bias is determined by the probability of obtaining an integer in the range $\{q, \ldots, 2^w - 1\}$, which is equal to $\frac{2^w - q}{2^w} \approx 2^{-4}$ for $q = 7681$. To reduce the bias in the obtained distribution, rejection sampling can be performed as proposed for instance in Kyber [12]. This requires generating a number of random integers $a_0, \ldots, a_r$ and selecting one that is in the interval $[0, q - 1]$. The sampling algorithm using r rejection steps, has a probability of returning an integer in the range $\{q, \ldots, 2^w - 1\}$ of approximately $2^{-4(1+r)}$. We implemented rejection sampling for $r = 0, 1, 2, 3$ and 4. The impact of the number of rejection steps on the area utilization is shown in Figure 14. There is a small area overhead in slices, LUTs and flipflops as rejection steps are added (the maximum for $r = 4$ is less than 3 percent). The number of clock cycles is not impacted by the additional r rejection steps for the range of r considered in Figure 14. The number of DSPs and BRAMs also remains the same. Rejection sampling can thus be efficiently implemented without much overhead.

B. Alternative PRNG

While Trivium leads to fast and small circuits, his 80-bit key space is smaller than the number of security bits (128, 192 or 256, depending on the parameter set) targeted by Kyber, NewHope and Frodo. An attacker has no direct access to the PRNG output used for error sampling. However, the correctness of a Trivium key guess can be checked by reconstructing $\mathbf{e}_1$ using the PRNG and verifying that $\mathbf{e}_1^\intercal \mathbf{A} \approx \mathbf{c}_1$. The Trivium key can therefore be found in 2^{80} operations. If the Trivium key is compromised, an attacker may compute $\mathbf{c}_2 - \mathbf{b}\mathbf{e}_1 \approx \lfloor \frac{q}{2} \rfloor \mu$ to recover the message. An exhaustive search in the 80-bit key space could thus be used for message recovery attacks.

In Kyber, NewHope and Frodo it is suggested to use SHAKE256 or similar algorithms as PRNG. Other schemes propose to use the less secure SHAKE128 to generate the public key part A. We implement a hybrid version using Trivium for the public key and SHAKE256 for the error samples, as presented in [18]. We also implement a version that uses SHAKE256 for both error sampling and public-key generation. A standalone Keccak implementation requires 1769 slices, 3782 LUTs and 5121 flipflops, and computes the 24-round Keccak algorithm in 25 clock cycles. The 3 variations (Trivium, Hybrid and SHAKE) are implemented for RLWE-1024, MLWE-512, MLWE-768 and MLWE-1024. The speed results are shown in Figure 15. The area overheads are shown in Figure 16 (number of required DSPs remains unchanged). Figure 17 presents a time×area trade-offs comparison. The hybrid version has a clear advantage over the full SHAKE256 variant in terms of area and speed. The obtained frequency in the RLWE implementation is heavily impacted by substituting Trivium for SHAKE, dropping from 256 MHz to 166 MHz. See Table VI for detailed results.

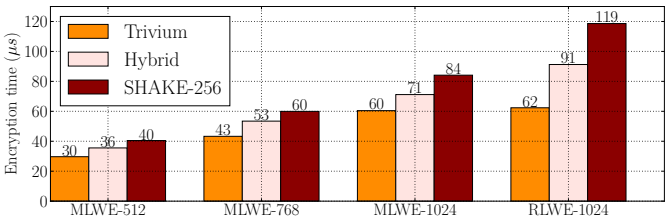


Fig. 15. Impact of the PRNG choice (Trivium, Hybrid, SHAKE) on the encryption time for CPA-secure MLWE and RLWE architectures.

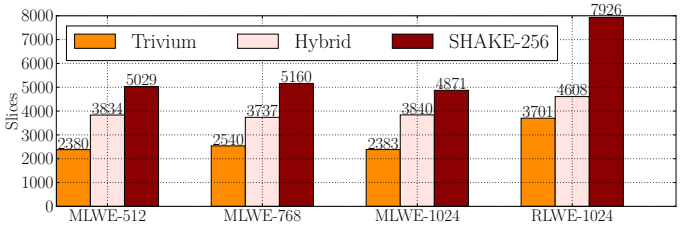


Fig. 16. Impact of the PRNG choice on area (in slices) for CPA-secure MLWE and RLWE architectures.

C. CCA Secure Solutions

As proposed for Frodo [10], NewHope [11] and Kyber [12], we transform our CPA-only secure LWE, RLWE and MLWE implementations (with hybrid sampling mode) into CCA secure implementations using algorithms in Figures 3 and 4. Hash functions H and G are instantiated with the SHA3-256 algorithm. A comparison between the timing results of the

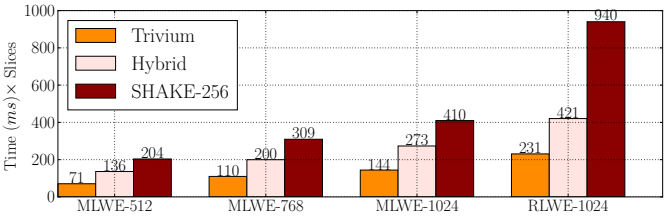


Fig. 17. Impact of the choice of PRNG on time × area for CPA-secure MLWE and RLWE architectures.

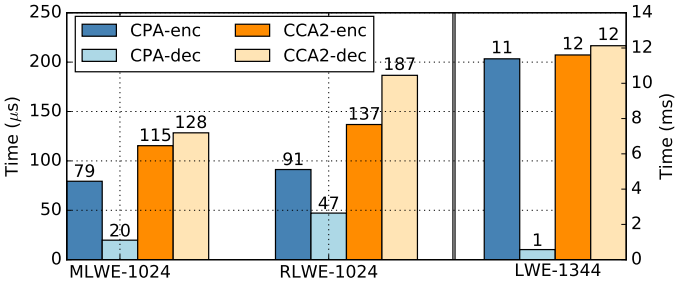


Fig. 18. Encryption and decryption time for CCA2 and CPA implementations. RLWE and MLWE times are in μs, while ms are used for LWE.

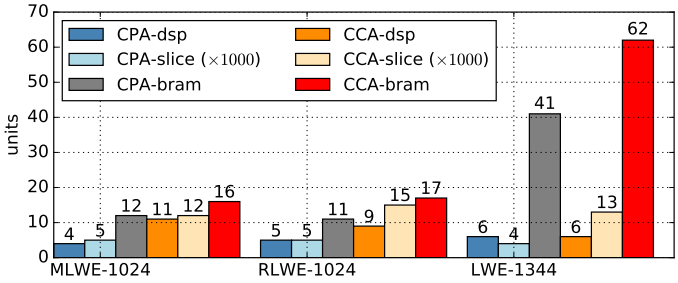


Fig. 19. Area comparison between CCA2 and CPA-only implementations.

CPA-only implementations and the CCA implementations is shown in Figure 18. For LWE, the computation time of the hash functions is small compared to the matrix multiplication. The computation time difference between CPA-secure and CCA-secure encryption are almost entirely accounted for by the matrix multiplication in the encryption algorithm. For MLWE and RLWE however, this is not the case. The difference in computation time between CPA-enc and CCA-enc is due to the hash functions, as can be expected from the encapsulation algorithm in Figure 3. This also holds for the difference between CCA-dec and the sum of CPA-enc and CPA-dec. Additional slow-down is caused by a drop in obtained frequency for CCA implementations. The impact on the area is shown in Figure 19. There is a clear increase in DSPs for RLWE and MLWE, showing that the sharing of resources is not optimal.

VIII. Comparison with other Works

Table VI summarizes results for encryption and encapsulation, including this work (TW) and the best FPGA implementations from the literature. State-of-the-art implementations use handwritten HDL to get optimized FPGA implementations (with more extensive implementation effort). All cited results have been implemented on Xilinx FPGAs (7 series and Ultrascale families), except the recent work [31] implemented on Intel/Altera FPGA, which makes direct comparisons difficult. Results in Table VI are first grouped by algorithms (R/M/LWE), second by size and then by type (CCA, CPA or KE). For each group, the results are reported by increasing execution time. }

MLWR and RLWR-based schemes are often chosen with a modulus q that is a power of 2 to ease modular reduction (see [17], [32]). . Another advantage is that there is no need for binomial error sampling, as the errors are generated by setting a number of LSBs to zero.

The work [33] provides hardware/software solutions for many PKE/KEM algorithms and sizes. The obtained results are interesting but cannot be compared to pure hardware ones (for instance the area required by a A53 Cortex core cannot be compared with FPGA resources).

For LWE, our implementation of the CCA secure LWE-640 takes more time (increased by a factor 2.45) but uses less DSP blocks (decreased by a factor 3.2) compared to the HDL optimized implementation [18] using hybrid PRNG. Our implementation uses more slices, but it includes decapsulation whereas [18] does not.

For RLWE, our results are given for $q = 7681$. Our CPA-secure RLWE-1024 implementation has an execution time very close to [34] (63μs vs. 62μs), but is less optimized in term of hardware resource consumption (4 vs. 2 DSPs). This shows some potential for HLS implementation.

For MLWE-1024 on Artix-7 FPGAs, our solution is slower (116μs vs. 67.9μs) and larger (factor 5) than [35] (published after our initial submission). Using the same HLS code, we are able to get a much faster solution on a more efficient FPGA (UltraScale+).

As presented in Fig. 15, changing from Hybrid PRNG to a complete SHAKE256 solution adds a time overhead about 30% for RLWE-1024 and 20% for MLWE-1024 due to the frequency drop. It also leads to 30% to 50% area increase. Thus, even with a complete SHAKE solution the results remain good using HLS.

Clearly, recent optimized HDL solutions outperform our HLS ones. But at the time of our initial submission, our HLS solutions were comparable, and sometimes better, to published results based on HDL implementation. Unfortunately, we are not able to compare the respective design efforts to implement many solutions for various algorithms, architectures and parameters. This would be interesting.

IX. Conclusion

We implemented several CPA and CCA secure LWE, RLWE and MLWE based cryptosystems on FPGA using HLS for the first time. At the submission time of this paper, our architectures generated using HLS lead to comparable, and sometimes better, results compared to the best references from the state of the art using HDL (such as [20], [19], [31], [25]) but probably for a much smaller design effort. Recent HDL based solutions (such as [36], [35]) are faster and smaller than ours. We showed how HLS can be used effectively to parallelize implementations. We also evaluated the impact of the choice of the PRNG on the performance of the encryption. Using Trivium instead of SHAKE to generate the pseudorandom part of the public key, the encryption can be accelerated. Even more speed-up is obtained when using Trivium for the error sampling as well, although this decreases the theoretical security of the scheme. HLS seems to us, for a first use of this type of hardware implementation method, an interesting solution for PQC algorithms in a tight design budget and for exploration of solutions. So many different implementations

TABLE VI
CPA AND CCA-SECURE ENCRYPTION OR ENCAPSULATION (CPA, CCA) OR 'CLIENT' PART IN SERVER-CLIENT-SERVER KEY EXCHANGE (K-E). "HYBRID" IN THE PRNG COLUMN MEANS TRIVIUM + SHAKE. NOTATIONS: SYMBOL * DENOTES RESULTS FOR BOTH ENCRYPTION AND DECRYPTION, TW STANDS FOR "THIS WORK".

Source	Lvl. of Parallel.	Scheme type-size	Algorithm	PRNG	Type	FPGA family (model)	Freq. MHz	Time μs	Area DSP, 18kb BRAM, Slices, LUT
[18]	high	LWE-640	Frodo	Hybrid	CCA	Artix-7	171	1212	16, 0, 1692, 5796
[18]	medium	LWE-640	Frodo	Hybrid	CCA	Artix-7	177	2342	8, 0, 1485, 5155
TW	low	LWE-640		Hybrid	CCA	Artix-7 (200)	159	2972	5, 37, 12951, 39077 *
[18]	low	LWE-640	Frodo	Hybrid	CCA	Artix-7	183	4624	4, 0, 1338, 4620
[25]		LWE-640	Frodo	Hybrid	CCA	Artix-7 (35T)	167	19608	1, 11, 1855, 6745
[18]	high	LWE-976	Frodo	Hybrid	CCA	Artix-7	168	2857	16, 0, 1782, 6188
[18]	medium	LWE-976	Frodo	Hybrid	CCA	Artix-7	175	5464	8, 0, 1608, 5562
TW	low	LWE-976		Hybrid	CCA	Artix-7 (200)	167	6317	14, 37, 13468, 41100 *
[18]	low	LWE-976	Frodo	Hybrid	CCA	Artix-7	180	10638	4, 0, 1455, 4996
[25]		LWE-976	Frodo	Hybrid	CCA	Artix-7 (35T)	166	45455	1, 16, 1985, 7209
TW	low	LWE-1344		Hybrid	CCA	Artix-7 200	167	11606	6, 62, 12299, 37342 *
[20]		RLWE-1024	NewHope	SHAKE	K-E	Zynq-7 (20)	131	79	8, 14, n.a. 20826
[19]		RLWE-1024	NewHope	SHAKE	K-E	Artix-7 (35T)	117	1532	2, 4, n.a., 4498
TW	medium	RLWE-512		Hybrid	CPA	Artix-7 (200)	211	50	5, 12, 7797, 16338 *
[34]		RLWE-1024	NewHope	SHAKE	CPA	Zynq-7 (20)	200	62	2, 8, n.a, 6781 *
TW	medium	RLWE-1024		Trivium	CPA	Artix-7 (200)	259	63	4, 10, 3701, 10112 *
[31]		RLWR-1018	Round5	SHAKE	CPA	Cyclone V (5csea5)	133	1000	4116 ALM, 10753 bytes *
TW	medium	RLWE-1024		Hybrid	CCA	Artix-7 200	167	137	9, 17, 14026, 42062 *
[31]		RLWR-1170	Round5	SHAKE	CCA	Cyclone V (5csea5)	130	1350	6337 ALM, 11765 bytes *
[36]		MLWR-512	Saber	SHAKE	CPA	UltraScale+	100	5.2	85, 12, n.a., 34886
[36]		MLWR-768	Saber	SHAKE	CPA	UltraScale+	100	11.6	85, 12, n.a., 34886
[36]		MLWR-1024	Saber	SHAKE	CPA	UltraScale+	100	21.0	85, 12, n.a., 34886
[35]		MLWE-512	Kyber	SHAKE	CCA	Artix-7 (12T)	161	30.5	2, 6, 2126, 7412
TW	medium	MLWE-512		Hybrid	CCA	Artix-7 (200)	170	60	11, 16, 11028, 34206 *
[37]	high	MLWR-768	Saber	SHAKE	CCA	UltraScale+ (9eg)	250	26	0, 2, n.a., 23600 *
[35]		MLWE-768	Kyber	SHAKE	CCA	Artix-7 (12T)	161	47.6	2, 6, 2126, 7412
TW	medium	MLWE-768		Hybrid	CCA	Artix-7 (200)	167	88	11, 16, 11890, 34145 *
[38]		MLWR-768	Saber	SHAKE	CCA	Zynq-7 (20)	125	4147	28, 4, n.a., 7400 *
TW	medium	MLWE-1024		Hybrid	CCA	UltraScale+ (9eg)	417	48	9, 16, 9314, 44964 *
[35]		MLWE-1024	Kyber	SHAKE	CCA	Artix-7 (12T)	161	67.9	2, 6, 2126, 7412
TW	medium	MLWE-1024		Hybrid	CCA	Artix-7 (200)	170	116	11, 16, 11567, 33707 *
[39]		MLWE-1024	Kyber	SHAKE	CCA	Artix-7 (35T)	60	6900	4, 34, n.a., 1738

TABLE VII
CCA-SECURE MLWE-1024 USING SHAKE256 FOR ERROR SAMPLING FOR DIFFERENT FPGA FAMILIES, USING VIVADO VERSION 2018.3.

FPGA family	Freq. MHz	Time μs enc/dec	Area DSP, BRAM, Slices, LUT
Artix-7	200	99/110	11, 16, 11322, 35607
Kintex-7	286	70/77	9, 16, 12066, 34175
Virtex-7	286	70/77	9, 16, 12508, 35718
Zynq UltraScale+	417	48/53	9, 16, 9314, 44964
Kintex UltraScale	333	61/68	9, 16, 7238, 43101
Virtex UltraScale	286	69/77	9, 16, 6474, 33979

and optimizations would have not been possible during a PhD thesis using HDL. We plan to implement other cryptosystems (e.g. isogenies, codes) using HLS and work on arithmetic support libraries for HLS.

ACKNOWLEDGMENT

This work has been supported by a PhD grant from PEC/DGA/Région Bretagne.

REFERENCES

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. Sci. Statist. Comput.*, vol. 26, p. 1484, 1997.

[2] O. Regev, "On Lattices, Learning With Errors, Random Linear Codes, and Cryptography," in *Proc. ACM Symposium on Theory of Computing*, May 2005, pp. 84–93.

[3] V. Lyubashevsky, C. Peikert, and O. Regev, "On Ideal Lattices and Learning with Errors over Rings," in *Proc. Conf. on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

[4] A. Langlois and D. Stehlé, "Worst-case to Average-case Reductions for Module Lattices," *Designs, Codes, and Cryptography*, vol. 75, no. 3, pp. 565–599, 2015.

[5] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Proc. Int. Conf. on the Theory and Applications of Cryptographic Techniques*, Cambridge, UK, 2012, pp. 719–737.

[6] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," NIST, Tech. Rep., 2016.

[7] E. Homsirikamol and K. G. George, "Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: The CAESAR contest case study," in *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 120–127.

[8] K. Basu, D. Soni, M. Nabeel, and R. Karri, "NIST post-quantum cryptography - a hardware evaluation study." *IACR Cryptol. ePrint Arch.*, p. 47, 2019.

[9] T. Zijlstra, K. Bigou, and A. Tisserand, "FPGA Implementation and Comparison of Protections against SCAs for RLWE," in *Proc. Int. Conf. on Cryptology in India (IndoCrypt)*, Dec. 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02309481

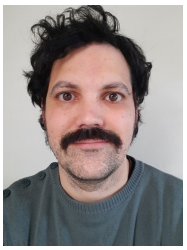
[10] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE," in *Proc. ACM Conf. on Computer and Communications Security*, Oct. 2016, pp. 1006–1018.

[11] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum Key Exchange - A New Hope," in *Proc. USENIX Security Symposium*, 2016, pp. 327–343.

[12] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM," in *Proc. IEEE European Symp. on Security and Privacy (EuroS&P)*, Apr. 2018, pp. 353–367.
[13] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," in *Proc. Int. Cryptology Conf.* Springer, 1999, pp. 537–554.
[14] C. De Cannière, "Trivium: A stream cipher construction inspired by block cipher design principles," in *Proc. Int. Conf. on Information Security*. Springer, 2006, pp. 171–186.
[15] M. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions," NIST FIPS 202, 2015.
[16] T. Zijlstra, K. Bigou, and A. Tisserand, "LWE crypto in HLS on FPGA," Online: https://sourcesup.renater.fr/www/lwe-hls-fpga/, 2020.
[17] J. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM," in *Proc. Int. Conf. Progress in Cryptology in Africa (AFRICACRYPT)*, May 2018, pp. 282–305.
[18] J. Howe, M. Martinoli, E. Oswald, and F. Regazzoni, "Optimised Lattice-Based Key Encapsulation in Hardware," in *Second PQC Standardization Conference*. NIST, 2019.
[19] T. Oder and T. Güneysu, "Implementing the NewHope-Simple key exchange on low-cost FPGAs," in *Proc. Conf. Cryptology and Information Security in Latin America (LATINCRYPT)*, 2017, pp. 128–142.
[20] P. Kuo, W. Li, Y. Chen, Y. Hsu, B. Peng, C. Cheng, and B. Yang, "Post-Quantum Key Exchange on FPGAs," *IACR Cryptology ePrint Archive*, p. 690, 2017.
[21] S. Roy, F. Vercauteren, N. Mentens, D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES)*, Sep. 2014, pp. 371–391.
[22] E. Ozcan and A. Aysu, "High-level-synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, pp. 1–1, 2019.
[23] T. Pöppelmann and T. Güneysu, "Area Optimization of Lightweight Lattice-Based Encryption on Reconfigurable Hardware," in *Proc. IEEE Int. Symp. on Circuits and Systemss (ISCAS)*, Jun. 2014, pp. 2796–2799.
[24] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized Schoolbook Polynomial Multiplication for Compact Lattice-Based Cryptography on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–5, 2019.
[25] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard Lattice-Based Key Encapsulation on Embedded Devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 3, pp. 372–393, aug 2018.
[26] J. A. Solinas, "Generalized Mersenne Numbers," Center for Applied Cryptographic Research, Univ. Waterloo, Tech. Rep. corr-99-39, 1999.
[27] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-kyber (version 2.0) – submission to round 2 of the NIST post-quantum project," Specification document (part of the submission package), Mar. 2019.
[28] "Vivado HLS optimization methodology guide UG1270 (v2017.4)," Xilinx, Dec. 2017.
[29] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *J. ACM*, vol. 15, no. 2, pp. 252–264, 1968.
[30] T. Pöppelmann, T. Oder, and T. Güneysu, "High-Performance Ideal Lattice-based Cryptography on 8-bit ATxmega Microcontrollers," in *Proc. Int. Conf. on Cryptology and Information Security in Latin America (LATINCRYPT)*, Aug. 2015, pp. 346–365.
[31] M. Andrzejczak, "The Low-Area FPGA Design for the Post-Quantum Cryptography Proposal Round5," in *Proc. Federated Conf. on Computer Science and Information Systems (FedCSIS)*. IEEE, 2019, pp. 213–219.
[32] H. Baan, S. Bhattacharya, S. Fluhrer, O. Garcia-Morchon, T. Laarhoven, R. Rietman, M. Saarinen, L. Tolhuizen, and Z. Zhang, "Round5: Compact and fast post-quantum public-key encryption," in *Proc. Int. Conf. on Post-Quantum Cryptography*. Springer, May 2019, pp. 83–102.
[33] V. B. Dang, F. Farahmand, M. Andrzejczak, and K. Gaj, "Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign," in *Proc. Int. Conf. on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 206–214.
[34] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT," *IACR Trans. on Cryptographic Hardware and Embedded Systems (TCHES)*, pp. 49–72, 2020.
[35] Y. Xing and S. Li, "A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 2, pp. 328–356, Feb. 2021.
[36] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "LWRpro: An energy-efficient configurable crypto-processor for module-LWR," *Transactions on Circuits and Systems I: Regular Papers*, vol. 68, pp. 1146–1159, Mar. 2021.
[37] S. Sinha Roy and A. Basso, "High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 4, pp. 443–466, Aug. 2020.
[38] J. M. B. Mera, F. Turan, A. Karmakar, S. S. Roy, and I. Verbauwhede, "Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism," *IACR crypto ePrint archive*, 2020.
[39] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic - accelerating Kyber and NewHope on RISC-V," Cryptology ePrint Archive, Report 049, 2020.
[40] C.-M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 2, pp. 159–188, Feb. 2021.



Timo Zijlstra was PhD student at CNRS and Lab-STICC laboratory at the time of writing and is now security evaluator at SERMA Safety & Security. His research interests are post quantum cryptography and side channel attacks.



Karim Bigou received the M. Sc. degree in cryptography in 2011 and the Ph. D. degree in computer science in 2014 from the University of Rennes 1. He is now associate professor at University of Western Brittany (UBO) in computer science in Lab-STICC laboratory. His research interests include computer arithmetic, applied cryptography, digital security, hardware and software implementations of cryptography.



Arnaud Tisserand is senior researcher at CNRS (French National Center for Scientific Research) in computer science in Lab-STICC laboratory. His research interests include computer arithmetic, computer architecture, digital security, VLSI and FPGA design, design automation, low-power design and applications in applied cryptography, scientific computing, digital signal processing.