FPGA Implementation and Comparison of Protections against SCAs for RLWE

Timo Zijlstra, Karim Bigou, Arnaud Tisserand

▶ **To cite this version:**

Timo Zijlstra, Karim Bigou, Arnaud Tisserand. FPGA Implementation and Comparison of Protections against SCAs for RLWE. 20th International Conference on Cryptology in India, Dec 2019, Hyderabad, India. hal-02309481

HAL Id: hal-02309481
https://hal.science/hal-02309481

Submitted on 9 Oct 2019

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.

FPGA Implementation and Comparison of Protections against SCAs for RLWE

Timo ZIJLSTRA[1], Karim BIGOU[2][0000−0001−9294−5594], and Arnaud TISSERAND[1][0000−0001−7042−3541]

[1] CNRS, Lab-STICC UMR 6285 and Université Bretagne Sud, Lorient, France
timo.zijlstra@univ-ubs.fr, arnaud.tisserand@univ-ubs.fr
[2] Université Bretagne Occidentale and Lab-STICC UMR 6285, Brest, France
karim.bigou@univ-brest.fr

Abstract. We present various FPGA implementations of protections against SCAs for RLWE-based PKE. We implemented the main solutions from the state of the art with improved variants. We also propose a new protection based on a redundant representation of the ring elements to randomize computations. We compare the implementation results of all these solutions.

Keywords: ring learning with errors, side channel attack, blinding, masking, shuffling, randomization

1 Introduction

The algorithms currently used in *public-key cryptography* (PKC) in most of applications are not secure against *quantum computers* using Shor's algorithm [28]. *Post-quantum cryptography* (PQC) relies on mathematical problems for which known quantum algorithms offer no significant speed-up. *Lattice* problems such as *learning with errors* (LWE) and *ring*-LWE (RLWE) received a lot of attention. A standardization project for post-quantum encryption and signatures has been launched by NIST in 2016 [8]. Its goal is to select and standardize post-quantum solutions to replace RSA and ECC. The second round started in January 2019, it includes 17 *public-key encryption* (PKE) submissions [1], and 9 of them are based on lattice problems.

While PQC is resistant against quantum computers, its implementation must be protected against *physical attacks*. *Side channel attacks* (SCAs) exploit the leakage of secret information through the analysis of the power consumption, electromagnetic radiation or computation timings of the cryptographic device. For instance, *correlation power analysis* (CPA) uses a set of *traces* obtained by measuring the power consumption of the device for different inputs.

The *secret key* in RLWE based cryptosystems consists of a polynomial in a finite ring. Decryption involves a multiplication with the secret polynomial. This multiplication is the ideal target for SCAs. One way to prevent such attacks is

to *randomize* the operands to remove the correlation between the power traces and the secret polynomial coefficients.

In this work, we implement in hardware various countermeasures from the state of the art against SCAs. We also improve some of them and propose a new one. We consider the *masking* scheme from [25] for which we propose a new masked decoder. Our decoder is deterministic and does not add to the decryption failure probability, as opposed to the one from [25]. We also implement two randomization techniques proposed by [27]: *blinding*, *shifting*; and a combination of the two. To the best of our knowledge, these are the first FPGA implementations of these techniques. We also propose a new countermeasure based on a *redundant representation* of the ring elements to randomize the computations during the decryption algorithm. Our new protection leads to a small overhead in terms of time and area. We also propose two methods to shuffle the operations during the point-wise multiplications. Finally, we compare all those solutions implemented on the same FPGA and using the same *high-level synthesis* (HLS) tools for fair comparison. HLS allows us to quickly evaluate several architectures and parameters. Our results show that HLS tools lead to implementations with similar, or even better, performances than VHDL or Verilog ones from the literature but for a significantly reduced design cost.

2 Definitions and Notations

- For q a prime number and n a power of two, let $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$.
- Lower case bold variables (*e.g.* $\mathbf{a}$) are polynomials in $\mathcal{R}_q$ of degree $< n$.
- $\mathcal{B}_\lambda$ denotes the symmetric binomial distribution centered around 0 with integer parameter λ.
- $\mathbf{a} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$ is a polynomial in $\mathcal{R}_q$ whose coefficients are sampled from $\mathcal{B}_\lambda$.
- $\odot$ is used for point-wise multiplication of polynomials and vectors.

3 State of the Art Analysis

3.1 Learning with Errors based PKE

The LWE problem and LWE based cryptography have been introduced by Regev in [24], where it is shown that LWE is at least as hard as some worst case lattice problems. The introduction of RLWE by [17] gives rise to more efficient cryptographic applications by adding an algebraic structure to the lattices. The matrices and vectors from Regev's cryptosystem are replaced by polynomials in finite rings, reducing the size of the public key and allowing fast multiplication algorithms. This speed-up has been studied by [11] in hardware implementation. The definition of RLWE we give here is a practical instantiation of the more general definition from [17].

Definition 1 (RLWE [17]). *For some (secret key) polynomial* $\mathbf{s} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$*, a RLWE sample is generated by sampling a polynomial* $\mathbf{a}$ *from the uniform distribution over* $\mathcal{R}_q$*, and sampling* $\mathbf{e} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$ *and computing the output* $(\mathbf{a}, \mathbf{b})$ *where* $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}$*. The search variant of the RLWE problem is to find* $\mathbf{s}$ *given a number of samples for* $\mathbf{s}$*.*

We describe the framework used for instance by NewHope. Other schemes may use deterministic errors ("Learning with Rounding") or Gaussian noise instead of using the binomial distribution. RLWE based submissions still present in the second round of the NIST standardization project include NewHope [2], LAC [15] and Round5 [4].

1. Key generation. Let $\mathbf{s} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$ be the private key. Sample a uniform random $\mathbf{a} \in \mathcal{R}_q$ and $\mathbf{e}_0 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$. The public key is given by the RLWE sample $(\mathbf{a}, \mathbf{b})$ where $\mathbf{b} := \mathbf{as} + \mathbf{e}_0$.
2. Encryption. Let the plaintext $\mathbf{m} \in \mathcal{R}_q$ be a polynomial with coefficients in the set $\{0, 1\}$ only. Sample 3 polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$. The ciphertext is given by $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_1 \leftarrow \mathbf{ae}_1 + \mathbf{e}_2$ and $\mathbf{c}_2 \leftarrow \mathbf{be}_1 + \mathbf{e}_3 + \lfloor \frac{q}{2} \rfloor \cdot \mathbf{m}$.
3. Decryption. Let $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}$. For each coefficient of $\mathbf{d}$, decode to 0 if it is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$, else decode to 1.

3.2 Components of the Cryptosystem

Encoding/Decoding. The maps between the message space $\{0, 1\}^n$ and $\mathcal{R}_q$ are called Encode and Decode. A string of bits is encoded by mapping 0 to 0 and 1 to $\lfloor \frac{q}{2} \rfloor$, resulting in a polynomial with coefficients in $\mathbb{Z}_q$. To decode a polynomial in $\mathcal{R}_q$, a coefficient is mapped to 0 if it is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$ in $\mathbb{Z}_q$, else it is mapped to 1. That is, if $c \in \left[\lfloor \frac{q}{4} \rfloor, \lfloor \frac{3q}{4} \rfloor \right)$ then $c \mapsto 1$, else $c \mapsto 0$.

Binomial sampling. The $\mathcal{B}_\lambda$ distribution over $\mathbb{Z}_q$ is sampled by generating 2λ uniformly random bits $a_1, \ldots, a_\lambda, b_1, \ldots, b_\lambda$ and returning $\sum_{i=1}^{\lambda}(a_i - b_i) \bmod q$.

Polynomial multiplication and NTT. The encryption and decryption functions both rely on polynomial multiplication in $\mathcal{R}_q$. The polynomial multiplication is a costly operation. Hardware implementations of RLWE schemes such as [20] tend to use the Number Theoretic Transform (NTT) to compute this operation. It has also been suggested to use the schoolbook algorithm for area optimization [21], but in general the NTT seems to yield better performance and highly area-optimized implementations exist [26].

To compute a polynomial multiplication using the NTT, the polynomials should be mapped to the NTT domain where the polynomial multiplication is a point-wise operation taking only n modular multiplications in $\mathbb{Z}_q$. Addition and subtraction can also be performed point-wise in the NTT domain. The inverse NTT is applied to bring the result back in the time domain.

Definition 2 (NTT). *Let $\omega \in \mathbb{Z}_q$ be a primitive n-th root of unity and $\mathbf{a}(x) = \sum_{i=0}^{n-1} a_i x^i$ an element in $\mathcal{R}_q$. Then the image of $\mathbf{a}$ under the NTT is given by $\hat{\mathbf{a}} = \sum_{j=0}^{n-1} \hat{a}_j x^j$, where $\hat{a}_j = \mathbf{a}(\omega^j)$.*

To use the NTT for multiplication in $\mathcal{R}_q$, the polynomials have to be pre-processed using the *negative wrapped convolution* (NWC) [16]. Let $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathcal{R}_q$ such that $\mathbf{a}(x)\mathbf{b}(x) = \mathbf{c}(x) + \mathbf{d}(x)(x^n + 1)$ in $\mathbb{Z}_q[x]$ for some $\mathbf{c}(x)$ of degree smaller than n. Let $\phi \in \mathbb{Z}_q$ be a primitive $2n$-th root of unity such that $\phi^n = -1 \bmod q$. Then, in $\mathbb{Z}_q[x]$ and for $i \geq 0$, one has:

$$\begin{aligned}\mathbf{a}(\phi\omega^i)\mathbf{b}(\phi\omega^i) &\equiv \mathbf{c}(\phi\omega^i) + \mathbf{d}(\phi\omega^i)((\phi\omega^i)^n + 1) \bmod q \\ &\equiv \mathbf{c}(\phi\omega^i) + \mathbf{d}(\phi\omega^i)(-1 + 1) \bmod q \\ &= \mathbf{c}(\phi\omega^i)\end{aligned}$$

This means that $\mathrm{NTT}(\mathbf{a}(\phi x)) \odot \mathrm{NTT}(\mathbf{b}(\phi x)) = \mathrm{NTT}(\mathbf{c}(\phi x))$. In other words, one gets the reduction $\bmod(x^n + 1)$ for free by applying the NTT to $\mathbf{a}(\phi x)$ instead of $\mathbf{a}(x)$. To obtain the correct result from the polynomial multiplication, the inverse of the NWC should be applied to $\mathrm{NTT}^{-1}(\mathrm{NTT}(\mathbf{c}))$. That is, each coefficient has to be multiplied by a power of ϕ^{-1}. Therefore, the values of $\phi^i \bmod q$ have to be precomputed for $0 < i < n$ and $-n < i < 0$.

The transform is efficiently computed in $\log_2(n)$ stages of n multiplications using the Cooley-Tukey algorithm [9]. Several optimizations have been proposed to accelerate this computation. The multiplication by the powers of the $2n$-th root of unity can be merged with the twiddle factors in the first stage or the scaling multiplication by $n^{-1} \bmod q$ [26]. Instead of precomputing n^{-1} and the powers of ϕ^{-1}, the values of $n^{-1}\phi^{-i}$ for $0 \leq i < n$ can be precomputed directly. A similar result merging the NWC with the final stage of the inverse NTT was described by [22]. By making clever use of the Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF) transforms, [22] shows that the bit reversal can be avoided. The DIT NTT is used to compute the inverse transformation. The DIT NTT takes an input in bit-reversed order and returns the output in the original order. The bit-reversal resulting from the DIF forward transformation is thus automatically undone by the inverse NTT. All the operations in the NTT domain are computed on the bit-reversed coefficient vectors. The public and private keys are therefore stored in bit-reversed order in the NTT domain. To limit the amount of modular reductions during the NTT, [14] allows variables to grow slightly larger than q.

Not counting symmetric primitives, the NTT is the most expensive operation in the scheme with a complexity of $\mathcal{O}(n \log n)$. To reduce the number of NTTs to be computed, the public and private keys can be stored in the NTT domain. The ciphertext part $\mathbf{c}_1$ must also be sent in the NTT domain. During the encryption, 2 forward NTTs and 1 inverse NTT have to be computed and during the decryption only 1 inverse NTT is needed.

Using the constant geometry variant [19] of the NTT algorithm, the memory access pattern is independent of the stage. The values are not read from the same memory as the one that the updated variables are written to, therefore 2

BRAMs are needed in the implementation. At each iteration of the stage loop, 2 values are read from the memory, a butterfly operation is computed and the 2 results are written to the memory. A detailed description of the stage is given by Algorithm 1. All arithmetic operations are performed in $\mathbb{Z}_q$.

Algorithm 1 i-th stage of the NTT [19]

1: **function** STAGE(X, i)
2: **for** $j \leftarrow 0$ to $\frac{n}{2} - 1$ **do**
3: $\theta \leftarrow \omega^{\lfloor \frac{j}{2^i} \rfloor \cdot 2^i}$ ▷ Get twiddle factor from memory
4: $(x_0, x_1) \leftarrow (X[2j], X[2j+1])$ ▷ Read from memory X
5: $\big(Y[j], Y[j+\frac{n}{2}]\big) \leftarrow (x_0 + x_1, (x_0 - x_1)\theta)$ ▷ Write to memory Y
6: **return** Y

Modular reduction. Modular reduction for moduli of the form $q = 2^{l_1} - 2^{l_2} + 1$ can be efficiently computed using algorithms in the style of [29]. 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.

3.3 Side Channel Analysis

Power analysis attacks on cryptographic implementations have first been described by [12]. Side channel attacks on LWE cryptography exploit vulnerabilities in Gaussian sampling algorithms [6], [10], polynomial multiplication [3] or the NTT [23]. The decryption algorithm makes use of the secret key and is therefore vulnerable to statistical and machine learning attacks on side channels such as power consumption (for instance differential power analysis, DPA) or electromagnetic radiations (differential electromagnetic analysis, DEMA).

3.4 Protections

All the operations in the decryption handle inputs that depend on the secret key. To protect it against DPA, these inputs should be randomized at the start of each decryption. The decryption algorithm decodes the coefficients of some polynomial $\mathbf{d}$ where $\mathbf{d}$ is defined by $\mathbf{d} = \mathbf{c}_2 - \mathrm{NTT}^{-1}(\mathbf{c}_1 \odot \mathbf{s})$. It should be noted that knowledge of the coefficients of $\mathbf{d}$ leads to complete key recovery in the chosen plaintext attack model. Since $\mathbf{c}_1$ and $\mathbf{c}_2$ are known inputs and $\mathbf{c}_1$ is invertible in $\mathcal{R}_q$ with high probability, one can compute $\mathbf{c}_1^{-1} \cdot (\mathbf{c}_2 - \mathbf{d}) = \mathbf{s}$. To prevent SCAs on the coefficients of $\mathbf{d}$, these coefficients should not be computed directly. Instead, a randomized or masked version of $\mathbf{d}$ is used. The decoder should therefore be able to decode randomized or masked inputs. We now present the main countermeasures from literature against statistical attacks on RLWE.

Masking. In [25] the secret key is split in two shares: $\mathbf{s} = \mathbf{s}' + \mathbf{s}''$ for some uniformly random $\mathbf{s}'$ at the start of each decryption. The linear part of the decryption function is computed twice: first the ciphertext is decrypted (but not decoded) using secret key $\mathbf{s}'$ and then using secret key $\mathbf{s}''$, yielding two polynomials $\mathbf{d}'$ and $\mathbf{d}''$. The final step consists of decoding the coefficients of $\mathbf{d} = \mathbf{d}' + \mathbf{d}''$ to bits. This is a non linear operation, that is, $\textsc{Decode}(a+b)$ is not necessarily equal to $\textsc{Decode}(a) + \textsc{Decode}(b)$. As an example, if $a = b = \lfloor \frac{q}{6} \rfloor$, $2 \times \textsc{Decode}\left(\lfloor \frac{q}{6} \rfloor\right) = 0$ but $\textsc{Decode}\left(2 \times \lfloor \frac{q}{6} \rfloor\right) = 1$. This means that one cannot simply apply the decoder to the coefficients of $\mathbf{d}'$ and $\mathbf{d}''$ separately and then add the results in $\mathbb{Z}_2$ to obtain the correct plaintext.

Because of the DPA scenario mentioned above, the two shares $\mathbf{d}'$ and $\mathbf{d}''$ should not be recombined before decoding to bits. Let d' and d'' denote a coefficient (of some fixed index) of polynomials $\mathbf{d}'$ and $\mathbf{d}''$ respectively. A *masked decoder* takes as input two coefficients $(d', d'') \in \mathbb{Z}_q^2$ and computes the value of $\textsc{Decode}(d' + d'')$ without computing $d' + d''$. The solution from [25] makes use of the fact that for some $(d', d'') \in \mathbb{Z}_q^2$ it is easy to deduce the value of $\textsc{Decode}(d' + d'')$. For instance, if $0 \leq d' < \frac{q}{4}$ and $\frac{q}{4} \leq d'' < \frac{q}{2}$ then it must hold that $\frac{q}{4} \leq (d' + d'') < \frac{3q}{4}$, therefore the coefficient decodes to 1. Similar "easy cases" exist, but not all $(d', d'') \in \mathbb{Z}_q^2$ can be resolved in this way. If both d' and d'' lie between 0 and $\frac{q}{4}$, all we know is that $0 \leq (d' + d'') < \frac{q}{2}$ and this can decode to either 0 or 1.

The idea from [25] to solve the hard cases is to *reshare* the two shares: for any $\delta \in \mathbb{Z}_q$ one has $d' + \delta + d'' - \delta = d' + d'' = d$. It is therefore possible to add any constant to one of the shares and subtract the same constant from the other one. However, there is no guarantee that $(d' + \delta, d'' - \delta)$ is an easy case. If the new shares still do not form an easy case, the shares have to be reshared again. In [25] a list of constants $\{\delta_1, \ldots, \delta_{16}\}$ is presented that is supposed to minimize the number of resharings to be performed. Their implementation refreshes the shares 16 times such that with high probability an easy case is obtained in at least one of the 16 iterations.

The computation time overhead due to the 16 iterations and the additional decoding failures are important drawbacks to this solution. [18] propose an alternative masked decoding. Their method effectively decodes without additional decoding failures. The comparison that they make between this decoder and their re-implementation of the one from [25] however shows only a very limited improvement in terms of performance. Their masked decryption takes over 3 times more cycles to compute than the unmasked version. The same implementation also uses a blinding countermeasure proposed by [27].

Blinding. With the blinding countermeasure [27] the polynomials $\mathbf{s}$ and $\mathbf{c}_1$ are multiplied by scalars a and b in $\mathbb{Z}_q$ respectively. The blinded polynomial multiplication is then computed: $(a\mathbf{s}) \cdot (b\mathbf{c}_1) = (ab)\mathbf{s} \cdot \mathbf{c}_1$. The inverse $(ab)^{-1}$ should be computed to obtain $\mathbf{s} \cdot \mathbf{c}_1$. [27] suggested to use (pre-computed) powers of ω and ω^{-1} as blinding factors to avoid the modular inversion. The decoding

process cannot be protected from DPA with the scalar blinding method. The blinding multiplication has to be inverted before the coefficients can be decoded.

Shifting. It is also suggested in [27] to apply a random anti-cyclic shift to the coefficients vector of the polynomials before multiplying. Due to the ring structure, this anti-cyclic shift corresponds to a multiplication by some power of x. For some random $i, j < n$, $(x^j \mathbf{s}(x)) \cdot (x^i \mathbf{c}_1(x)) = x^{i+j} \mathbf{s}(x) \mathbf{c}_1(x)$ is computed and $\mathbf{s}(x)\mathbf{c}_1(x)$ can be recovered by inversing the shift.

In practice it is not possible to obtain $x^i \mathbf{s}$ and $x^j \mathbf{c}_1$ by applying anti-cyclic shifts to their coefficients vectors, because they are represented in the NTT domain. To multiply by x^i in the NTT domain, observe that

$$\mathrm{NTT}(x^i) = (1, \omega^i, \omega^{2i}, \ldots, \omega^{(n-1)i}), \tag{1}$$

and $\mathrm{NTT}((\phi x)^i) = \phi^i \cdot \mathrm{NTT}(x^i)$. All of the coefficients of $\mathrm{NTT}(x^i)$ are already pre-computed, since they are exactly the n powers of ω. Multiplication by x^i can thus be done by a pointwise multiplication with the powers of ω and ϕ^i (for the NWC). This multiplication has to be performed in bit-reversed order, since $\mathbf{s}$ and $\mathbf{c}_1$ are in the NTT domain.

Shuffling. Masking, blinding and shifting offer little to no protection against single trace attacks. The single trace attack by [23] exploits leakage from the operations performed during the NTT. In that paper, it is suggested to counter the attack by randomizing the order in which the butterfly operations are computed. During each stage, the $\frac{n}{2}$ butterfly operations can be computed in a random order. The same shuffling methods can also be applied to all the pointwise operations in the decryption.

4 Unprotected FPGA Implementation of RLWE

In this section, we present our implementation of the encryption and decryption algorithms described in the previous section on an Artix XC7A200 FPGA using Vivado HLS (version 2018.1). Our decryption architecture is the basis for the protected implementations proposed in the next sections. We compare our unprotected RLWE implementations with results from literature. One of our goals is to show that competitive results can be obtained using HLS from C code for a reduced design cost compared to VHDL or Verilog design.

Figure 1 presents the high-level architecture of our *accelerator*. For encryption, the public keys are first loaded into the local *RAM*, then the computations are performed by the *functional units* (FUs, see below). During encryption/decryption our accelerator is isolated for security reasons, it does not take any input or generate any output. After encryption/decryption, the result is sent out through the interface. In the paper, all the communications through the interface are included in our results. Depending on parameter n, the typical time spent for interfacing represents about 12% to 21% of the total encryption/decryption time.

Fig. 1. High level architecture of our accelerator.

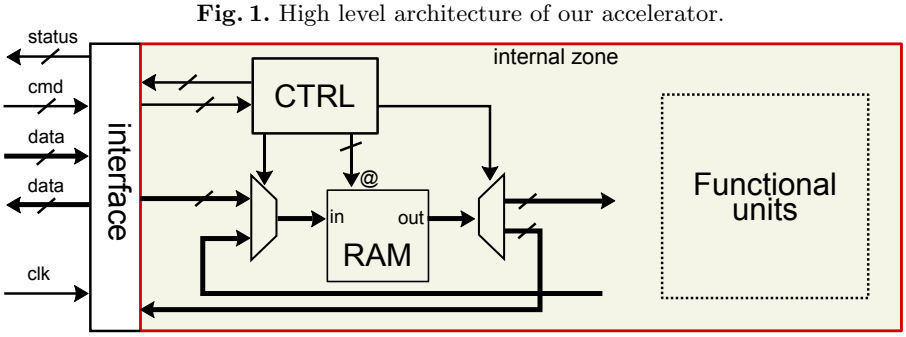
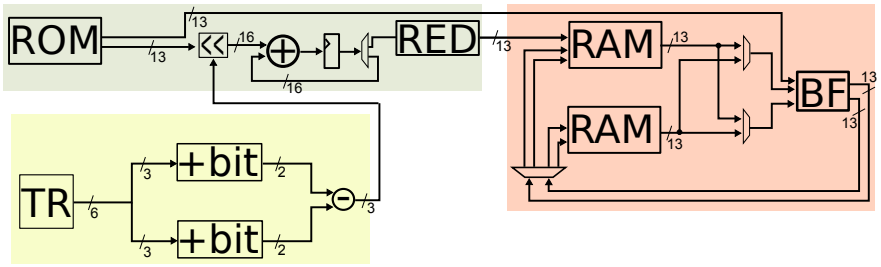


Fig. 2. Architecture computing the error polynomials in the NTT domain. The yellow part uses the PRNG (TR unit) to generate samples from the $\mathcal{B}_\lambda$ distribution (here: $\lambda = 3$). The +bit operator computes the sum of λ input bits. The NWC (upper left) is computed using a shift-based multiplier and modular reduction (RED). The NTT is computed on the right, using one Gentlemen-Sande butterfly (BF) operator.



Parameters. In order to compare with literature results, we implement RLWE for the parameter sets $(n, q, \lambda) = (1024, 12289, 8)$ and $(256, 7681, 3)$. For $n = 1024$, we use a simplification of the CPA-secure version of NewHope1024 PKE with key reuse. We do not implement the key refreshing, ciphertext compression/decompression and key encoding/decoding in this paper. For simplicity we use the Trivium stream cipher as PRNG.

Encryption and Decryption. Following [22], we avoid the bit-reversal step by implementing both the DIF and DIT NTT. The stage loop is fully pipelined, such that it takes just over $\frac{n}{2}$ clock cycles to complete one stage. The complete forward transformation is computed in few more than $\frac{n}{2} \log n$ cycles.

The error polynomials $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$ are sampled from the binomial distribution $\mathcal{B}_\lambda(\mathcal{R}_q)$. The required random bits are provided by the PRNG. Since the ciphertext part $\mathbf{c_1} = \mathbf{a}\mathbf{e}_1 + \mathbf{e}_2$ will be sent in the NTT domain, the NTT has to be applied to both $\mathbf{e}_1$ and $\mathbf{e}_2$. The NWC must be computed for both polynomials. To compute these multiplications, we use the fact that the coefficients are sampled from $\mathcal{B}_\lambda$ and therefore are bounded by $-\lambda$ and λ. The multiplications can be computed using only a few shifts and additions, without using a DSP block.

Table 1. FPGA implementation results for our RLWE solutions (denoted V1, V2 and V3) and literature solutions. If specified, Encryption/Decryption and Server/Client/Server (for a 3-step key exchange) timing results are shown separately. Separate area results for Server and Client are indicated with +.

Source	FPGA	Latency (clock cycles)	MHz	Time (μs)	Slice, DSP, LUT, BRAM
$n = 256$					
[20]	XC6VLX75	6861/4404	262	26.2/16.8	1506, 1, 4549, 12
[26]	XC6VLX75	6300/2800	313	20.1/9.1	n.a., 1, 1349, 2
V1	XC7A200	5039/2188	208	24.2/10.5	1624, 1, 4365, 5
V2	XC7A200	3764/2239	250	15.1/9.0	2122, 6, 5616, 8
$n = 1024$					
[13]	XC7Z020	6900/10300/2800	133/131	51.9/78.6/21.1	n.a., (8+8), (18756+20826), (14+14)
[30]	XC7A035	171124/179292	125/117	1369/1532	0, (2+2), (5142+4498), (4+4)
V3	XC7A200	16146/9586	250	64.6/38.3	4106, 7, 11164, 12

The NTT is then applied to $\mathbf{e}_1$ and $\mathbf{e}_2$ simultaneously, using two parallel NTT units each consisting of one butterfly unit and two BRAMs. The architecture for sampling $\mathbf{e}_1$ (or $\mathbf{e}_2$) and mapping it to the NTT domain is shown in Figure 2.

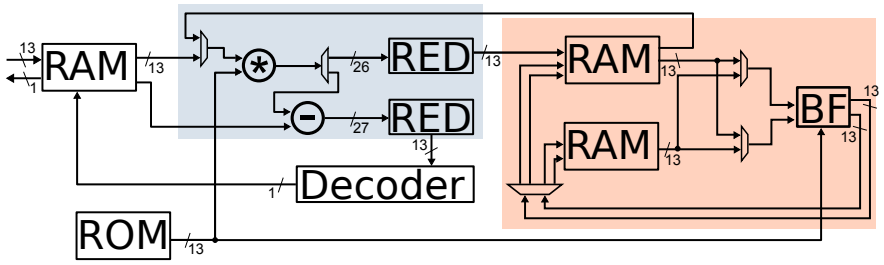
The architecture for decryption is shown in Figure 3. The area and timing implementation results for RLWE are shown in Table 1 with similar solutions from the literature.

Our small implementation is denoted by V1. This implementation with only 1 DSP block is comparable in size and speed to [20] but is larger and 15 to 20% slower than the cryptoprocessor from [26]. By computing the forward NTTs in parallel in V2, we are faster than both, but more DSP blocks are needed. For $n = 1024$, the key exchange implementation by [13] is comparable with our V3 results in terms of speed, but the V3 implementation uses 50% less DSP blocks and BRAMs. We conclude that results obtained using HLS are comparable or, in the best case, even better in terms of speed (up to 25%) and/or area (up to 50%) than results from works based on VHDL or Verilog implementations.

5 New Variants of State of the Art Protections

The protections proposed in this section and the next one are implemented by modifying our base architecture from Figure 3 for $n = 256$ and $q = 7681$. In real-world applications these protections should be part of an architecture implementing the CCA2-secure version of the scheme, including a re-encryption of the decrypted ciphertext and several evaluations of some hash function.

Fig. 3. Architecture for the decryption. The ciphertext is completely loaded in the RAM before starting the computations. The two pointwise operations (one before and one after the inverse NTT) in the blue region share a DSP block.



5.1 Masking with a New Masked Decoder

We implement a variant of the masking scheme described in the state of the art [25], improving the masked decoding process. We propose a simple masked decoder that does not need 16 iterations and that does not increase the decoding failure probability. Let $d', d'' \in \left[0, \frac{q}{4}\right)$, then $d' + d'' \in \left[0, \frac{q}{2}\right)$. If either d' or d'' were to be shifted by exactly the right amount (cf. 4), then we would be able to determine if either $d' + d'' \in \left(\frac{-q}{4}, \frac{q}{4}\right)$ or $d' + d'' \in \left(\frac{q}{4}, \frac{3q}{4}\right)$. The trick is then to find a $\delta \in \left[\frac{-q}{4}, \frac{q}{4}\right]$ such that $d' + \delta$ changes quadrant while $d'' - \delta$ does not (or the other way around). Suppose that $d' \geq d''$ and let $\delta = 1 + \min(\left\lfloor \frac{q}{4} \right\rfloor - d', d'')$. Then, depending on the value of δ, there are two possibilities for the new shares $d' + \delta$ and $d'' - \delta$:

1. $d' + \delta = \left\lfloor \frac{q}{4} \right\rfloor + 1 \in \left[\frac{q}{4}, \frac{q}{2}\right)$ and $d'' - \delta$ is in the same interval as d''. Then $d' + d''$ must be in the interval $\left(\frac{q}{4}, \frac{3q}{4}\right)$, therefore we decode to 1.
2. $d' + \delta$ is in the same interval as d' and $d'' - \delta = -1 \in \left[\frac{-q}{4}, 0\right)$. Then $d' + d''$ must be in the interval $\left[0, \frac{q}{4}\right) \cup \left(\frac{-q}{4}, 0\right]$ and therefore we decode to 0.

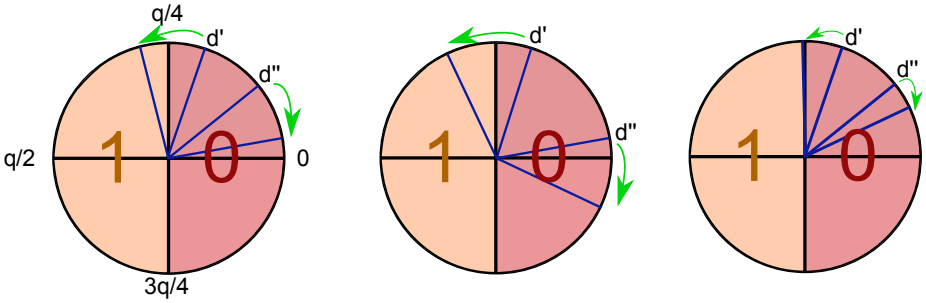
Similar solutions can be found for the other hard cases. Let Q_i denote the interval $\left[\frac{iq}{4}, \frac{(i+1)q}{4}\right)$ for $0 \leq i \leq 3$, that we will refer to as "quadrants". The property that allows to solve the easy cases is the following:

Property 1. If $d' \in Q_i$ and $d'' \in Q_j$ then $(d' + d'') \in Q_{i+j \bmod 4} \cup Q_{i+j+1 \bmod 4}$.

In the remainder of this section, we let i and j be the quadrant indices of d' and d'' respectively. For $i + j = 1 \bmod 4$ it follows from Property 1 that $(d' + d'') \in Q_1 \cup Q_2$. In other words, the sum lies in the left half of $\mathbb{Z}_q$ and therefore decodes to 1. Similarly, the $(d', d'') \in \mathbb{Z}_q^2$ for which $i + j = 3 \bmod 4$ are easy cases and decode to 0.

The hard cases are given by $(d', d'') \in \mathbb{Z}_q^2$ for which $i + j = 0 \bmod 4$ or $i + j = 2 \bmod 4$, that is, $(d' + d'') \in Q_0 \cup Q_1$ or $(d' + d'') \in Q_2 \cup Q_3$ respectively. To reduce to an easy case, it suffices to move either (but not both) d' or d'' to an adjacent quadrant. Then for the new pair $(d' + \delta, d'' - \delta)$ exactly 1 mod 4 is

Fig. 4. Left: given that $d', d'' \in \left[0, \frac{q}{4}\right)$, there is no simple way to determine if $d' + d'' \in \left(\frac{q}{4}, \frac{3q}{4}\right)$, *i.e.* this is a hard case. Adding some δ to d' while subtracting the same δ from d'' yields a new pair (d', d'') which is an easy case. Middle: for some hard cases, adding and subtracting a constant δ does not give a solution: the new pair (d', d'') is another hard case. Right: d' is closer to $\frac{q}{4}$ than d'' is to 0. We therefore let $\delta = 1 + \left\lfloor \frac{q}{4} \right\rfloor - d'$. Then by construction, $d' + \delta$ changes quadrant while $d'' - \delta$ does not. It follows that $d = d' + \delta + d'' - \delta > \frac{q}{4}$ and $d < \frac{3q}{4}$. Therefore (d', d'') decodes to 1.



added to or subtracted from the sum $i + j$. Then for the updated i, j it holds that $i + j = 1 \bmod 4$ or $i + j = 3 \bmod 4$ and Property 1 applies.

It is always possible to modify the sum $i + j$ for the i, j corresponding to the shares by exactly 1. Assume w.l.o.g. that $d' \geq d''$. If $d' \in Q_i$, $d'' \in Q_j$ and d' is closer to $\frac{iq}{4}$ than d'' is to $\frac{(j-1)q}{4}$, then there is a δ such that $d' + \delta \in Q_{i+1}$ and $d'' - \delta \in Q_j$. If the opposite holds, then d'' can be moved to Q_{j-1} by subtracting a δ while d' stays in Q_i. The new pair $(d' + \delta, d'' - \delta)$ forms an easy case. This method does not work when the distance δ'' between d'' and $\frac{(j-1)q}{4}$ is equal to the distance δ' between d' and $\frac{iq}{4}$. However, these are exactly the cases for which $d' + d''$ is equal to either $\left\lfloor \frac{q}{4} \right\rfloor$ or $\left\lfloor \frac{3q}{4} \right\rfloor$. This means that even an unmasked decoder would not be able to decode these cases correctly. The parameters in LWE-based cryptoschemes are usually chosen such that such cases appear with negligible probability.

The comparison operation $\delta' < \delta''$ has to be implemented with caution. Generally, comparisons are performed by checking the bit sign of the subtraction of its operands. Since $\delta' - \delta'' = -(d' + d'') + \left\lfloor \frac{kq}{4} \right\rfloor$ for some integer $k < 4$, this operation leaks information about the unmasked value of d.
Instead of implementing a combinatory circuit, we have implemented successive accesses to a look-up table to perform the comparison. The implemented algorithm is described in Algorithm 2, where the bits of a and b are denoted $(a_0, \ldots, a_{w-1})$ and $(b_0, \ldots, b_{w-1})$ respectively. The look-up table implements the function T defined by $T(a_i, b_i, Y) = \big(a_i \wedge (\overline{b_i} \vee Y)\big) \vee (\overline{b_i} \wedge Y)$.

Note that it is not necessary to assume that $d' > d''$. Given (d', d'') and their corresponding quadrant indices $(i, j) = index(d', d'')$, the distances $\delta' = \left\lfloor \frac{(i+1)q}{4} \right\rfloor - d'$ and $\delta'' = d'' - \left\lfloor \frac{jq}{4} \right\rfloor$ are computed and compared. We have that:

Algorithm 2 Returns *True* if and only if $a > b$

```
1: function COMPARE(a, b)
2:     Y ← False
3:     for i = 0 to w − 1 do
           Y ← T(a_i, b_i, Y)
4:     return Y
```

$$\begin{aligned}\delta' < \delta'' &\iff \left\lfloor \frac{(i+1)q}{4} \right\rfloor - d' < d'' - \left\lfloor \frac{jq}{4} \right\rfloor \\ &\iff \left\lfloor \frac{(j+1)q}{4} \right\rfloor - d'' < d' - \left\lfloor \frac{iq}{4} \right\rfloor,\end{aligned}$$

which means that swapping d' and d'' (and their corresponding indices) does not change the boolean outcome of the comparison. The complete masked decoder is given by Algorithm 3. The new reshared parts $d' + \delta$ and $d'' - \delta$ do not need to be computed explicitely. The comparison of δ' and δ'' yields sufficient information to update the indices (i, j) and determine the decoded bit.

Algorithm 3 Proposed masked decoder for (d', d'')

```
1: function DECODE(d', d'')
2:     r ←$ {0, 1}                                  ▷ Mask for output
3:     (i, j) ← index(d', d'')                      ▷ Quadrant indices
4:     if i + j ≡ 1 mod 4 then
5:         return (r, r̄)                            ▷ Easy cases i + j ≡ 1 or 3.
6:     else if i + j ≡ 3 mod 4 then
7:         return (r, r)
8:     else
9:         δ' ← ⌊((i+1)q)/4⌋ − d'                   ▷ Distance to interval boundaries
10:        δ'' ← d'' − ⌊jq/4⌋
11:        if COMPARE(δ'', δ') then
12:            if i + j + 1 ≡ 1 mod 4 then
13:                return (r, r̄)
14:            else
15:                return (r, r)
16:        else
17:            if i + j − 1 ≡ 1 mod 4 then
18:                return (r, r̄)
19:            else
20:                return (r, r)
```

In order to make this masked decoder compatible with CCA2-secure implementations, the output is also masked. Instead of returning the plaintext bit, a

random bit is generated and XORed with the unmasked decoding result. The decoder returns both the mask and the masked value.

A total of $2^{n \log q} = 2^{3328}$ different masks can be obtained. The security of the masking scheme with its original decoder is experimentally evaluated by [25]. They also mention the (small) possibility of horizontal DPA attacks targeting the 16 iterations of their masked decoder. Our proposed decoder does not have this vulnerability since it does not use 16 iterations.

5.2 Shifting

In [27] there is no mention of any masked decoder. To secure the complete decryption function, we propose to apply the (normal) decoder to the shifted polynomial $x^{i+j}\mathbf{c}_2(x) - x^{i+j}\mathbf{s}(x)\mathbf{c}_1(x)$, meaning that $\mathbf{c}_2(x)$ should be shifted separately. The plaintext can then be obtained by applying the inverse shift to the decoded polynomial. The minus sign that comes with the anti-cyclic shift does not change the value of the decoded coefficient, because $\forall a \in \mathbb{Z}/q\mathbb{Z}$ it holds that $\textsc{Decode}(a) = \textsc{Decode}(-a)$. The decryption procedure for a ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$ can then be described as follows:

1. Generate random $i, j < n$.
2. Compute $\mathrm{NTT}(x^i) \odot \mathbf{s}$ and $\mathrm{NTT}(x^j) \odot \mathbf{c}_1$ by multiplying $\mathbf{s}$ and $\mathbf{c}_1$ by the powers of ω and ϕ in an order determined by i and j respectively.
3. Compute the pointwise product to obtain $x^{i+j}\mathbf{s} \cdot \mathbf{c}_1$ and apply in the inverse NTT.
4. Apply the anti-cyclic $(i+j)$- shift to $\mathbf{c}_2$ and obtain $x^{i+j}\mathbf{c}_2$.
5. Compute the subtraction $x^{i+j}\mathbf{c}_2 - x^{i+j}\mathbf{s} \cdot \mathbf{c}_1 = x^{i+j}(\mathbf{c}_2 - \mathbf{s} \cdot \mathbf{c}_1)$.
6. Decode to obtain the shifted plaintext. Shift $i + j$ positions to the left.

5.3 Blinding

The blinding countermeasure is implemented by generating two random indices $0 \leq i, j < n$ and multiplying $\mathbf{c}_1$ and $\mathbf{s}$ by ω^i and ω^j respectively.

5.4 Shifting and Blinding combined

Both shifting and blinding involve multiplication by the powers of ω and ϕ. To shift the polynomial $\mathbf{s}(x)$ by $i < n$ positions, we compute $\phi^i \cdot \mathrm{NTT}(x^i) \odot \mathbf{s}(x)$. With almost no additional costs, this operation can be combined with the blinding operation by simply modifying the exponents of ω. To shift the polynomial by i positions and blind using ω^{-j} for some $j < n$, we use:

$$\omega^{-j}\phi^i \cdot \mathrm{NTT}(x^i) \odot \mathbf{s}(x) = (\phi^i\omega^{-j}, \phi^i\omega^{i-j}, \ldots, \phi^i\omega^{(n-1)i-j}) \odot \mathbf{s}(x) \qquad (2)$$

Both $\mathbf{s}$ and $\mathbf{c}_1$ are shifted and blinded. The combined blinding factor has to be removed before the decoding. The combination of the two countermeasures is therefore somewhat more expensive than shifting alone. The decoding is performed in the shifted order.

Both shifting and blinding use two $\log(n)$-bit randomization factors. As pointed out by [27], the total amount of added noise entropy for shifting and blinding combined is $4\log(n)$ bits. For $n = 256$ this is equal to 32.

6 New Protections

6.1 Shuffling

The first of the two shuffling methods proposed in this paper consists of replacing loop counters by linear feedback shift registers (LFSR). An LFSR is parametrized by an irreducible polynomial $f \in \mathbb{F}_2[x]$ and its degree k. It computes $x^i a \bmod f$ for $0 \leq i < 2^k - 1$ and some initial state $a \in \mathbb{F}_2[x]/f$. The computed values are exactly all the $2^k - 1$ invertible elements of the finite field $\mathbb{F}_2[x]/f$. The order in which they are computed is determined by the initial state a. Multiplication by x in $\mathbb{F}_2[x]/f$ is very fast and can be computed using only 1 shift and a XOR on bit positions depending on f. Our second shuffling method consists of generating a random permutation using a permutation network in the style of [5].

LFSR method. Let an LFSR be parametrized by some irreducible polynomial f of degree $\frac{n}{2}$. We let $k = \log_2(n) - 1$ and consider the coefficients vectors of polynomials in $\mathbb{F}_2[x]/f$ to be the binary representations of integers ranging from 0 to $\frac{n}{2} - 1$. The LFSR thus generates a sequence of $\frac{n}{2} - 1$ integers that will serve as indices for the loop counter in Algorithm 1. Instead of computing the i-th butterfly operation at the i-th loop iteration, we compute the butterfly operation that is on the j-th position, where j is the index corresponding to the i-th element generated by the LFSR. In other words, the normal loop counter is replaced by an LFSR. The LFSR has only $2^k - 1$ outputs, whereas we need 2^k for the $\frac{n}{2}$ butterfly operations. Therefore the first operation of each stage is not shuffled: it is always computed in the first loop iteration of the stage.

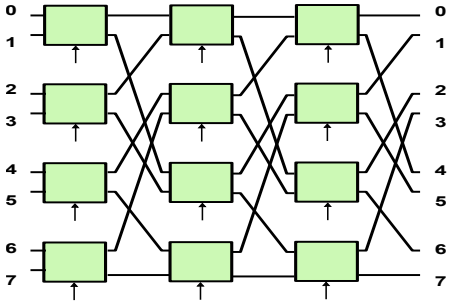
To obtain a meaningful permutation, we use the PRNG to generate a new initial state a at the start of each stage. Since $a = 0$ is not allowed as initial state, we set $a = 1$ as the default state in the case that the PRNG outputs 0. The initial state is thus set to default state with probability $\frac{4}{n}$. All the other initial states appear with probability $\frac{2}{n}$. This slight bias could be reduced by having the PRNG generate multiple initial states and selecting a non-zero state.

The $2^k - 1$ possible initial states determine $2^k - 1$ unique sequences. The operations of a complete $\log_2(n)$ stage NTT can then be computed in $(2^k - 1)^{\log_2(n)}$ different ways. For $n = 256$ and $k = 7$ this is more than 2^{55}. With the LFSR method applied to the pointwise operations outside the NTT as well, the total number of random bits added is equal to 71. A single trace attack like [23] that requires all of the $\log_2(n)$ stages seems unlikely to succeed on an implementation using the LFSR countermeasure as described.

We use an LFSR of degree $k = \log_2(n)$ in a similar manner to shuffle the order of the n pointwise multiplications outside the NTT.

Drawbacks to the LFSR loop counter include a limited permutation space, a slightly biased outcome and the fact that the first element is not permuted.

Fig. 5. Permutation network for $N = 8$. Each box is controlled by a random bit and swaps the inputs if this bit is equal to 1.



Permutation Network Method. We propose to use a permutation network generator in the style of [5]. Their permutation generator is designed for use in AES and is impractical for larger ($N = 256$) permutations. It is also biased. We simplify their permutation network to obtain a permutation generator that can generate $N^{N/2}$ permutations and that is uniform on its range. In the remainder of this section, the parameter N is the size of the permutation, which is equal to n for the shuffling of the pointwise operations. To shuffle the butterfly operations during the computation of the NTT, N is substituted by $\frac{n}{2}$.

For $b \in \{0, 1\}$, let the operators $T_b : \{0, \ldots, N-1\} \to \{0, \ldots, N-1\}$ be defined by the mapping $x \mapsto \left\lfloor \frac{x}{2} \right\rfloor + b\frac{N}{2}$. Then T_0 is a bitwise shift erasing the least significant bit (LSB), and T_1 applies the same shift and sets the MSB to 1.

The permutation network consists of $k = \log_2(N)$ stages and takes as input $(x_0, \ldots, x_{N-1}) = (0, \ldots, N-1)$. During each stage, $\frac{N}{2}$ random bits $b_1, \ldots, b_{N/2}$ are generated and for all pairs (x_{2i}, x_{2i+1}) the images $T_{b_i}(x_{2i})$ and $T_{\bar{b}_i}(x_{2i+1})$ are computed. In other words, for each pair (x_{2i}, x_{2i+1}), one is sent to position i, while the other is mapped to $i + \frac{N}{2}$. This is equivalent to writing one bit of the image of x_{2i} under the generated permutation and writing its complement to the image of x_{2i+1}. The network is shown for $N = 8$ in Figure 5. It is exactly the same as the computation scheme of the constant geometry NTT, in which the butterfly operators are replaced by controlled swap operators.

For any integer $0 \leq x < N$ the image of x under the generated permutation can be written as $T_{b_1} \circ \cdots \circ T_{b_k}(x)$ and is equal to the value corresponding to the binary representation $(b_1, \ldots, b_k)_2$. The $\frac{kN}{2}$ control bits thus determine the image of each index under the generated permutation. By uniqueness of binary representation it follows that any modification to any subset of the $\frac{kN}{2}$ control bits would modify the generated permutation as well. The permutation generator is therefore an injective map from $\{0, 1\}^{Nk/2}$ into the set of all permutations Σ_N. This means that the number of possible configurations of the $\frac{kN}{2}$ control bits, which is equal to $N^{N/2}$, is exactly the number of permutations that can be generated by the network. The number of different permutations that can be obtained is 2^{1024} for the pointwise operations and 2^{256} for the NTT. Moreover,

the output of the permutation network is uniform on its range given uniformly random input.

Since the permutation space is much larger than the one we obtain with the LFSR, we will only generate one $\{0, \ldots, \frac{n}{2} - 1\} \rightarrow \{0, \ldots, \frac{n}{2} - 1\}$ permutation for the NTT at the start of each decryption. Each stage is then computed in the order defined by this permutation. We also compute only one permutation of size n that will be used for all the pointwise operations during one decryption.

6.2 Randomization using Redundant Number Representation

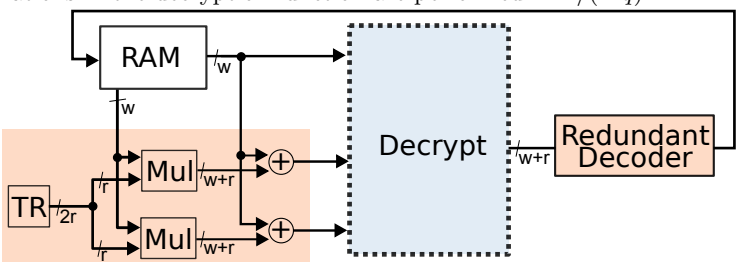
In RSA and ECC, some exponent or scalar randomization countermeasures have been proposed against SCAs (see for instance [7]). A secret exponent or scalar can be randomized without loss of information by adding a random multiple of the group order to it. The corresponding power traces are thus randomized, removing correlation between the side channel traces and the secret key. A similar concept can be applied to RLWE.

We can add random multiples of the modulus q to the secret key coefficients without invalidating the secret key. This is done at the start of each decryption. The PRNG is used to generate small r-bit random numbers for some integer parameter r. These numbers are multiplied by q and then added to the input and to the secret key. We then continue using arithmetic operations in $\mathbb{Z}/(2^r q)\mathbb{Z}$ instead of in $\mathbb{Z}_q$. The fact that for all $a, b \in \mathbb{Z}$ we have that $ab \bmod (2^r q) \equiv ab \bmod q$, ensures us that the result is in the correct equivalence class.

The redundancy is not removed for the decoding. Instead we modify the algorithm to decode the coefficients directly from $\mathbb{Z}/(2^r q)\mathbb{Z}$ to $\{0, 1\}$. The new decoder returns 0 if the input lies in the union of sets $\bigcup_{i=0}^{2^r} \left[\frac{-q}{4} + iq, \frac{q}{4} + iq\right)$ and returns 1 if the input is in $\bigcup_{i=0}^{2^r} \left[\frac{q}{4} + iq, \frac{3q}{4} + iq\right)$.

At each execution of the algorithm, the multipliers, adders and decoder are handling different inputs. The computations (and the corresponding traces) are thus randomized. A total of $256r$ random bits are added to the operands.

Fig. 6. Architecture with our redundant representation countermeasure. Before the decryption, small r-bit random multiples of q are added to the coefficients of $\mathbf{c}_1$ and $\mathbf{s}$. The operations in the decryption function are performed in $\mathbb{Z}/(2^r q)\mathbb{Z}$.



Validation through Correlation Power Analysis Simulations. We evaluate the robustness of our countermeasure based on a redundant representation by simulating CPAs. The polynomial multiplication in the NTT domain consists of n independent multiplications in $\mathbb{Z}/q\mathbb{Z}$. They are of the form $c \cdot s \bmod q$, where s is a coefficient of the secret key and c is a coefficient of the input ciphertext. We simulate correlation attacks on one modular multiplication of a known input coefficient c with an unknown secret key coefficient s.

We assume that the attacker observes the modular multiplication $c \cdot s \bmod q$ for a number of different (known) inputs c. For each modular multiplication she/he obtains the exact Hamming weight (HW) of the result. The attacker computes the "predictions": the HW of the value $c \cdot s \bmod q$ for all subkey candidates $s \in \mathbb{Z}_q$ and for all inputs c. She/he evaluates the correlation between the observed HW and the predictions. For each subkey possibility $\tilde{s} \in \mathbb{Z}_q$, the Pearson's correlation coefficient between the observed HW and the predictions is computed. Without countermeasures, the highest correlation is obtained for the correct subkey guess.

The inputs are randomized by adding a multiple of q and used in computations in $\mathbb{Z}/(2^r q)\mathbb{Z}$ for some redundancy parameter r. The impact of our countermeasure on the effectiveness of the CPA can be seen (for $q = 7681$) in Figure 7. Without redundancy ($r = 0$), the attacker observes the exact HW of the value $c \cdot s \bmod q$ for different values of c. These HWs coincide with the predictions for the correct subkey guess, resulting in a correlation coefficient of 1. For higher levels of redundancy, the average of the correlation coefficient for the correct subkey guess decreases.

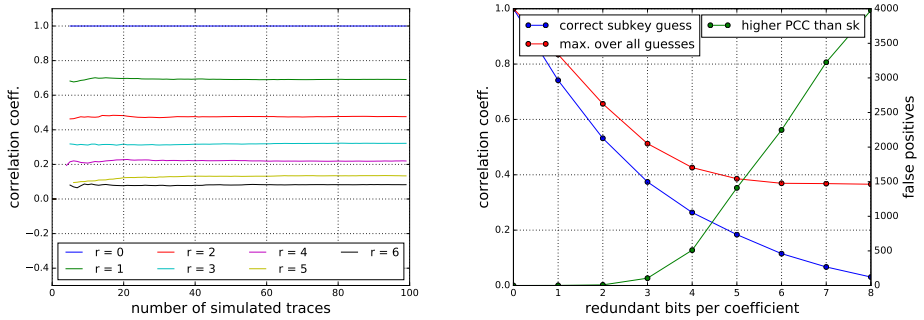
The right side of Figure 7 shows that the maximum correlation is obtained for incorrect subkey guesses for all $r \geq 1$. We refer to subkey guesses that yield to a higher correlation coefficient than the correct subkey guess as *false positives*. The number of these false positives increases with the redundancy level. For $r = 8$ and $r = 9$ there are on average around $\frac{q}{2}$ subkey guesses, for one coefficient of the secret key, that yield to a higher correlation coefficient than the correct key guess. Our countermeasure ensures that an exponential number of up to $\left(\frac{q}{2}\right)^n$ guesses have to be tested to recover the complete secret key.

7 Comparison of all Protections

FPGA implementation results for RLWE solutions with various countermeasures are presented in Table 2. Results from [25] are reported, and we also reimplemented their solution on an Artix-7 XC7A200 (denoted "A7") to provide fair comparisons. We also implemented the blinding and shifting methods from [27] and our shuffling methods. To the best of our knowledge, these are the first FPGA implementations for these countermeasures. Finally, the results for masking with our new masked decoder and our redundant randomized countermeasures are reported. The amount of randomness added for each countermeasure is specified in the second column of the table.

\renewcommand4pt3pt

Fig. 7. Mean correlation over 1000 simulations between the correct subkey guess and the observed HW as a function of the number of traces (left) and the number of redundant bits per coefficient (right). Right: average (1000 simulations of 100 traces each) of the maximum correlation over all subkey guesses is shown in red and the number of subkey guesses with higher correlation than the correct secret key in green.



We cannot directly compare our re-implementation of the masking from [25] and their original results on a Virtex-II XC2VP7 (denoted "V2"). However, it can be seen that the impact of masking on the performance of their V2 implementation is very high compared to our A7 re-implementation. The computation time for decryption is tripled. This is probably because the number of arithmetic operations in $\mathbb{Z}_q$ is doubled while no parallelism is used. Moreover, it seems that their masked decoder is implemented sequentially. In our re-implementation of the masked decoder from [25], we use parallelism to significantly reduce the performance penalty of their 16-step decoder. This increases the area.

Our new masked decoder is relatively simple and requires a small area (about 20% reduction compared to the re-implementation of the decoder from [25]), with almost the performance of the unprotected implementation. Compared to the unprotected solution, we use extra DSP blocks and BRAMs to compute the decryptions of the two shares in parallel.

The blinding implementation gives a slightly slower solution. Its area overhead is smaller than for both masking techniques. However, we stress that this blinding countermeasure should be used in combination with another countermeasure (as specified in [18]), since the blinding factor is removed before the decoding step. The shifting implementation yields to similar overhead (although with lower frequency) and its combination with blinding seems to be worthwhile. The permutation network is relatively costly in area. The LFSR loop counter is cheaper and slightly faster.

Finally, our redundant randomized countermeasure does not need additional DSPs or BRAMs to be implemented for small redundancy parameters ($r \leq 4$) and can therefore be used as a cheap way to secure the decryption. For higher redundancy levels the multiplication cannot be computed within a single 18×25

Table 2. FPGA results for RLWE with various countermeasures and $(q, n) = (7681, 256)$. The source column refers to the work in which the countermeasure was first proposed in LWE context. Timing and area results are for the decryption only.

Counter-measure	Entropy added (bits)	Src.	Impl.	FPGA	Lat.	Clk. (ns)	Time (μs)	Slice, LUT, DSP, BRAM
None	0	-	[25]	V2	2800	8.3	23.5	-, 1713, 1, -
Masking	3328	[25]	[25]		7500	10	75.2	-, 2014, 1, -
None	0	-	this work	A7	2357	3.3	7.8	$483, 1163, 2, 3$
Blinding	16	[27]			2768	3.8	10.6	$941, 2284, 3, 4$
Shifting	16	[27]			3138	4.7	14.8	$832, 2150, 3, 4$
Shift + Blind	32	[27]			3183	4.6	14.7	$1063, 2781, 3, 4$
Masking	3328	[25]			2517	4.0	10.1	$2187, 5500, 5, 6$
Our Mask.	3328	this work			2510	4.0	10.1	$1722, 4269, 5, 6$
Permutation	1280				2521	4.5	11.4	$3183, 7385, 2, 4$
LFSR ctr.	71				2846	3.6	10.3	$1069, 2861, 2, 3$
$r = 1$	256				2272	3.7	8.5	629, 1599, 2, 3
$r = 2$	512				2273	3.6	8.2	611, 1664, 2, 3
$r = 3$	768				2333	3.8	8.9	807, 2067, 2, 3
$r = 4$	1024				2338	3.6	8.5	872, 2285, 2, 3
$r = 5$	1280				2352	3.8	9.0	990, 2677, 2, 6
$r = 6$	1536				2394	3.9	9.4	1254, 3466, 3, 6
$r = 7$	1792				2410	3.9	9.4	1713, 5017, 3, 6
$r = 8$	2048				2426	3.9	9.5	2544, 7837, 3, 6

bits multiplier, as the ones hardwired in the Artix DSP blocks. A few additional DSP blocks and BRAMs are needed.

8 Conclusion

In this work, we compared several countermeasures against SCAs for RLWE from [25], [27] and proposed new ones. Our first proposed countermeasure is an adaptation of [25] with a new masked decoder which is deterministic. Our second one uses a redundant representation to randomize polynomial coefficients. We also implemented two different methods for shuffling. All the countermeasures (from literature and our ones) have been implemented on FPGA to evaluate the overhead compared to a common reference implementation on the same FPGA. Our new decoder uses over 20% less slices and LUTs than the one from [25]. To the best of our knowledge, we also present the first FPGA implementations for the blinding and shifting countermeasures from [27], and a combination of the two. Finally, our protection based on redundancy at ring level provides a cheap randomization method with an adjustable security/overhead trade-off.

In the future, we will explore other types of architectures, operators, algorithms and countermeasures (*e.g.* at architecture level). We also plan to use our solutions in the context of application benchmarks and evaluate their security against SCAs using a hardware setup under development in our research group.

Acknowledgment

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

References

1. G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Quynh Dang, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and Yi-Kai Liu. Status report on the first round of the NIST post-quantum cryptography standardization process. Technical report, 2019.
2. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A New Hope. In *Proc. 25th USENIX Security Symposium*, pages 327–343, 2016.
3. A. Aysu, Y. Tobah, M. Tiwari, A. Gerstlauer, and M. Orshansky. Horizontal side-channel vulnerabilities of post-quantum key exchange protocols. In *Proc. IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 81–88, May 2018.
4. H. Baan, S. Bhattacharya, S. R. Fluhrer, Ó. García-Morchón, T. Laarhoven, R. Rietman, M.J.O. Saarinen, L. Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In *10th International Conference on Post-Quantum Cryptography (PQCrypto)*, pages 83–102, 2019.
5. A.G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4):20:1–20:19, January 2012.
6. L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In *Proc. 18th International Conference on Cryptographic Hardware and Embedded Systems(CHES)*, pages 323–345, August 2016.
7. T. Chabrier and A. Tisserand. On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations. In *Proc. 21st Symposium on Computer Arithmetic (ARITH)*, pages 219–228. IEEE Computer Society, April 2013.
8. L. Chen, S. Jordan, Yi-Kai Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. Technical report, 2016.
9. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
10. T. Espitau, P.-A. Fouque, B. Gérard, and M. Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1857–1874, November 2017.
11. N. Göttert, T. Feller, M. Schneider, J. A. Buchmann, and S. A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Proc. 14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 512–529, September 2012.
12. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. 19th Annual International Cryptology Conference (CRYPTO)*, pages 388–397, August 1999.
13. Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. Post-quantum key exchange on FPGAs. *IACR Cryptology ePrint Archive*, 2017:690, 2017.

14. P. Longa and M. Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Proc. 15th International Conference on Cryptology and Network Security (CANS)*, pages 124–139, November 2016.
15. Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. LAC: practical ring-LWE based public-key encryption with byte-level modulus. *IACR Cryptology ePrint Archive*, 2018:1009, 2018.
16. V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A modest proposal for FFT hashing. In *Proc. 15th International Workshop on Fast Software Encryption (FSE)*, pages 54–72, February 2008.
17. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proc. 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 1–23. June 2010.
18. T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical CCA2-secure and masked ring-LWE implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(1):142–174, 2018.
19. M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *J. ACM*, 15(2):252–264, 1968.
20. T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *Proc. 20th International Conference on Selected Areas in Cryptography (SAC)*, pages 68–85, August 2013.
21. T. Pöppelmann and T. Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *Proc. IEEE International Symposium on Circuits and Systemss (ISCAS)*, pages 2796–2799, June 2014.
22. T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *Proc. 4th International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 346–365, August 2015.
23. R. Primas, P. Pessl, and S. Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *Proc. 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 513–533, September 2017.
24. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proc. 37th Annual ACM Symposium on Theory of Computing*, pages 84–93, May 2005.
25. O. Reparaz, S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. A masked ring-LWE implementation. In *Proc. 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 683–702, September 2015.
26. S. Sinha Roy, F. Vercauteren, N. Mentens, D. Donglong Chen, and I. Verbauwhede. Compact Ring-LWE cryptoprocessor. In *Proc. 16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 371–391, September 2014.
27. M.-J. O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures - engineering a side-channel resistant post-quantum signature scheme with compact signatures. *J. Cryptographic Engineering*, 8(1):71–84, 2018.
28. P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997.
29. J. A. Solinas. Generalized mersenne numbers. Technical Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, 1999.
30. T. Güneysu T. Oder. Implementing the NewHope-Simple key exchange on low-cost FPGAs. In *Proc. 5th International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, September 2017.