Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware

Thomas Pöppelmann and Tim Güneysu

Horst Görtz Institute for IT-Security, Ruhr University Bochum, Germany

Abstract. With this work we provide further evidence that lattice-based cryptography is a promising and efficient alternative to secure embedded applications. So far it is known for solid security reductions but implementations of specific instances have often been reported to be too complex beyond any practicability. In this work, we present an efficient and scalable micro-code engine for Ring-LWE encryption that combines polynomial multiplication based on the Number Theoretic Transform (NTT), polynomial addition, subtraction, and Gaussian sampling in a single unit. This unit can encrypt and decrypt a block in 26.19 µs and 16.80 µs on a Virtex-6 LX75T FPGA, respectively – at moderate resource requirements of about 1506 slices and a few block RAMs. Additionally, we provide solutions for several practical issues with Ring-LWE encryption, including the reduction of ciphertext expansion, error rate and constant-time operation. We hope that this contribution helps to pave the way for the deployment of ideal lattice-based encryption in future real-world systems.

Keywords: Ideal Lattices, Ring-LWE, FPGA Implementation

1 Introduction and Motivation

Resistance against quantum computers and long term security has been an issue that cryptographers are trying so solve for some time [12]. However, while quite a few alternative schemes and problem classes are available, not many of them received the attention both from cryptanalysts and implementers that would be needed to establish the confidence and efficiency for their deployment in real-world systems. In the field of patent-free lattice-based public-key encryption there are a few promising proposals such as a provably secure NTRU variant [47] or the cryptosystem based on the (Ring) LWE problem [29, 33]. For the latter scheme Göttert et al. presented a proof-of-concept implementation in [19] demonstrating that LWE encryption is feasible in software. However, their corresponding hardware implementation is quite large and can only be placed fully on a Virtex-7 2000T and does not even fit onto the largest Xilinx Virtex-6 FPGA for secure parameters.[1] Several other important aspects for Ring-LWE encryp-

[1] The authors report that the utilization of LUTs required for LWE encryption exceeds the number of available LUTs on a Virtex-6 LX240T by 197% and 410% for parameters $n = 256$ and $n = 512$, respectively. Note that the Virtex-6 LX240T is a very expensive €1300 FPGA (as of May 2013) and the largest of the Virtex-6 family.

tion have also not been regarded yet, such as the reduction of the extensive ciphertext expansion and constant-time operation to withstand timing attacks.

Contribution. In this work we aim to resolve the aforementioned deficiencies and present an efficient hardware implementation of Ring-LWE encryption that can be placed even on a low-cost Xilinx Spartan-6 FPGA. Our implementation of Ring-LWE encryption achieves significant performance, namely 42.88 µs to encrypt and 27.51 µs to decrypt a block, even with very moderate resource requirements on the low-cost Spartan-6 family. Providing the evidence that Ring-LWE encryption can be both fast and cheap in hardware, we hope to complement the work by Göttert et al. [19] and demonstrate that lattice-based cryptography is indeed a promising and practical alternative for asymmetric encryption in future real-world systems. In summary, the contributions of this work are as follows:

1. *Efficient hardware implementation of Ring-LWE encryption.* We present a micro-code processor implementing Ring-LWE encryption as proposed by [29, 33] in hardware, capable to perform the Number Theoretic Transform (NTT), polynomial additions and subtractions as well as Gaussian sampling. For a fair comparison of our implementation with previous work, we use the same parameters as in [19] and improve their results by at least an order of magnitude considering throughput/area on a similar reconfigurable platform. Moreover, our processor is designed as a versatile building block for the implementation of future ideal lattice-based schemes and is not solely limited to Ring-LWE encryption. All parts of our implementation have constant runtime and inherently provide resistance against timing attacks.
2. *Efficient Gaussian sampling.* We present a constant-time Gaussian sampler implementing the inverse transform method. The sampler is optimized for sampling from narrow Gaussian distributions and is the first hardware implementation of this method in the context of lattice-based cryptography.
3. *Reducing ciphertext expansion and decryption failure rates.* A major drawback of Ring-LWE encryption is the large expansion of the ciphertext[2] and the occurrence of (rare) decryption errors. We analyze different approaches to reduce the impact of both problems and harden Ring-LWE encryption for deployment in real-world systems.

In order to allow third-party evaluation of our results we will make source code files, test-benches and documentation available on our website.[3]

Outline. In Section 2 we introduce the implemented ring-based encryption scheme. The implementation of our processor, the Gaussian sampler and the cryptosystem are discussed in Section 3. In Section 4 we give detailed results including a comparison with previous and related works and conclude with Section 5.

[2] For example, the parameters used for implementation in [19] result in a ciphertext expansion by a factor of 26.

[3] See our web page at `http://www.sha.rub.de/research/projects/lattice/`

2 The Ring-LWEEncrypt Cryptosystem

In this section we briefly introduce the original definition of the implemented Ring-LWE public key encryption system (RING-LWEENCRYPT) and propose modifications in order to decrease ciphertext expansion and error rate without affecting the security properties of the scheme.

2.1 Background on LWE

Since the seminal result by Ajtai [2] who proved a worst-case to average-case reduction between several lattice problems, the whole field of lattice-based cryptography has received significant attention. The reasons for this seems to be that the underlying lattice problems are very versatile and allow the construction of hierarchical identity based encryption (HIBE) [1] or homomorphic encryption [17, 39] but have also led to the introduction of reasonably efficient public-key encryption systems [19, 29, 33], signature schemes [13, 20, 31], and even hash functions [32]. A significant long-term advantage of such schemes is that quantum algorithms do not seem to yield significant improvements over classical ones and that some schemes exhibit a security reduction that relates the hardness of breaking the scheme to the presumably intractable problem of solving a worst-case (ideal) lattice problem. This is a huge advantage to heuristic and patent-protected schemes like NTRU [26], which are just related to lattice problems but might suffer from yet not known weaknesses and had to repeatedly raise their parameters as immediate reaction to attacks [25]. A particular example is the NTRU signature scheme NTRUSign which has been completely broken [15, 40]. As a consequence, while NTRU with larger parameters can be considered secure, it seems to be worthwhile to investigate possible alternatives.

However, the biggest practical problem of lattice-based cryptography are huge key sizes and also quite inefficient matrix-vector and matrix-matrix arithmetic. This led to the definition of cyclic [37] and more generalized ideal lattices [30] which correspond to ideals in the ring $\mathbb{Z}[x]/\langle f \rangle$ for some irreducible polynomial f of degree n. While certain properties can be established for various rings, in most cases the ring $R = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1 \rangle$ is used. Some papers proposing parameters then also follow the methodology to choose n as a power of two and q a prime such that $q \equiv 1 \bmod 2n$ and thus support asymptotic quasi-linear runtime by direct usage of FFT techniques. Recent work also suggests that q does not have to be prime in order to allow security reductions [11].

Nowadays, the most popular average-case problem to base lattice-based cryptography on is presumably the learning with errors (LWE) problem [46]. In order to solve the decisional Ring-LWE problem in the ring $R = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1 \rangle$, an attacker has to decide whether the samples $(a_1, t_1), \ldots, (a_m, t_m) \in R \times R$ are chosen uniformly random or whether each $t_i = a_i s + e_i$ with $s, e_1, \ldots, e_m$ has small coefficients from a Gaussian distribution D_σ [33].[4] This distribution D_σ is

[4] Note that this is the definition of Ring-LWE in `Hermite normal form` where the secret s is sampled from the noise distribution D_σ instead of uniformly random [34].

defined as the (one-dimensional) discrete Gaussian distribution on $\mathbb{Z}$ with standard deviation σ and mean 0. The probability of sampling $x \in \mathbb{Z}$ is $\rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$ where $\rho_\sigma(x) = \exp\left(\frac{-x^2}{2\sigma^2}\right)$ and $\rho_\sigma\ (\mathbb{Z}) = \sum_{k=-\infty}^{\infty} \rho_\sigma(k)$. In this simple case the standard deviation σ completely describes the Gaussian distribution. Note that some works, e.g., [19,29] use the parameter $s = \sqrt{2\pi}\sigma$ to describe the Gaussian.

2.2 Ring-LWEEncrypt

The properties of the Ring-LWE problem can be used to realize a semantically secure public key encryption scheme with a reduction to decisional Ring-LWE. The scheme has been introduced in the full version [34] of Lyubashevsky et al. [33] and parameters have been proposed by Lindner and Peikert [29] as well as Göttert et al. [19]. The scheme (`Gen`, `Enc`, `Dec`) is defined as follows and will from now on be referred to as RING-LWEENCRYPT:

– `Gen`(a): Choose $r_1, r_2 \leftarrow D_\sigma$ and let $p = r_1 - a \cdot r_2 \in R$. The public key is p and the secret key is r_2 while r_1 is just noise and not needed anymore after key generation. The value $a \in R$ can be defined as global constant or chosen uniformly random during key generation.
– `Enc`$(a, p, m \in \{0,1\}^n)$: Choose the noise terms $e_1, e_2, e_3 \leftarrow D_\sigma$. Let $\bar{m} =$ `encode`$(m) \in R$, and compute the ciphertext $[c_1 = a \cdot e_1 + e_2, c_2 = p \cdot e_1 + e_3 + \bar{m}] \in R^2$
– `Dec`$(c = [c_1, c_2], r_2)$: Output `decode`$(c_1 \cdot r_2 + c_2) \in \{0,1\}^n$.

During encryption the encoded message $\bar{m}$ is added to $pe_1 + e_3$ which is uniformly random and thus hides the message. Decryption is only possible with knowledge of r_2 since otherwise the large term ae_1r_2 cannot be eliminated when computing $c_1r_2 + c_2$. According to [29] the polynomial a can be chosen during key generation (as part of each public key) or regarded as a global constant and should then be generated from a public verifiable random generator (e.g., using a binary interpretation of π). The encoding of the message of length n is necessary as the noise given by $e_1r_1 + e_2r_2 + e_3$ is still present after calculating $c_1r_2 + c_2$ and would prohibit the retrieval of the binary message after decryption. Note that the noise is relatively small as all noise terms are sampled from a narrow Gaussian distribution. With the simple threshold encoding `encode`$(m) = \frac{q-1}{2}m$ the value $\frac{q-1}{2}$ is assigned only to each binary one of the string m. The corresponding decoding function needs to test whether a received coefficient $z \in [0..q-1]$ is in the interval $\frac{q-1}{4} \leq z < 3\frac{q-1}{4}$ which is interpreted as one and zero otherwise. As a consequence, the maximum error added to each coefficient must not be larger that $|\frac{q}{4}|$ in order to decrypt correctly. The probability of an decryption error is mainly dominated by the tailcut and the standard deviation of the Gaussian $\sigma = \frac{s}{\sqrt{2\pi}}$. Decreasing s decreases the error probability but also negatively affects the security of the scheme.

Parameter Selection. For details regarding parameter selection we refer to the work by Lindner and Peikert [29] who propose the parameter sets (n, q, s) with

$(192, 4093, 8.87), (256, 4093, 8.35)$, and $(320, 4093, 8.00)$ for low, medium, and high security levels, respectively. In this context, Lindner and Peikert [29] state that medium security should be roughly considered equivalent to the security of the symmetric AES-128 block cipher as the decoding attack requires an estimated runtime of approximately 2^{120} seconds for the best runtime/advantage ratio. However, they did not provide bit-security results due to the new nature of the problem and several trade-offs in their attack.

In this context, the authors of [19] introduced hardware-friendly parameter sets for medium $(256, 7681, 11.31)$ and high security $(512, 12289, 12.18)$. With n being a power of two and q a prime such that $q = 1 \bmod 2n$, the Fast Fourier Transform (FFT) in $\mathbb{Z}_q$ (namely the Number Theoretic Transform (NTT)) can be directly applied for polynomial multiplication with a quasi-linear runtime of $\mathcal{O}(n \log n)$. Increased security parameters (e.g., a larger n) have therefore much less impact on the efficiency compared to other schemes [33].

Security Implications of Gaussian Sampling. For practical and efficiency reasons it is common to bound the tail of the Gaussian. As an example, the authors of the first proof-of-concept implementation of Ring-LWEEncrypt [19] have chosen to bound their sampler to $[-\lceil 2s \rceil, \lceil 2s \rceil]$. Unfortunately, they do not provide either a security analysis or justification for this specific value. In this context, the probability of sampling ± 24 which is out of this bound (recall that $\lceil 2s \rceil = \lceil 2 \cdot 11.32 \rceil = 23$) is $6.505 \cdot 10^{-8}$ and thus not negligible. However, when increasing the tail-cut up to a certain level it can be ensured that certain values will only occur with a negligible probability. For $[-48, 48]$, the probability of sampling an $x = \pm 49$ is $2.4092 \cdot 10^{-27} < 2^{-80}$ which is unlikely to happen in a real world scenario. The overall quality of a Gaussian random number generator (GRNG) can be measured by computing the statistical distance $\Delta(X, Y) = \frac{1}{2} \sum_{\omega \in \Omega} |X(\omega) - Y(\omega)|$ over a finite domain Ω between the probability of sampling a value x by the GRNG and the probability given by $\rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$.

Since in general attacks on LWE work better for smaller secrets (see [3, 4] for a survey on current attacks) the tail-cut will certainly influence the security level of the scheme. However, we are not aware of any detailed analysis whether short tails or certain statistical distances lead to better attacks. Moreover, a recent trend in lattice-based cryptography is to move away from Gaussian to very small uniform distributions (e.g., $-1/0/1$) [20, 38]. It is therefore not clear whether a sampler has to have a statistical distance of 2^{-80} or 2^{-100} (which is required for a worst-case to average-case reductions) in order to withstand practical attacks. Moreover the parameter choices for the Ring-LWEEncrypt scheme and for most other practical lattice-based schemes already sacrifice the worst-case to average-case reduction in order to obtain practical parameters (i.e., small keys). As a consequence, we primarily implemented a $\pm\lceil 2s \rceil$ bound sampler for straightforward comparison with the work by Göttert et al. [19] but also provide details and implementation results for larger sampler instantiations that support a much larger tail.

2.3 Improving Efficiency

In this section we propose efficient modifications to Ring-LWEEncrypt to decrease the undesirable ciphertext expansion and the error rate at the same level of security.

Reducing the Ciphertext Expansion. Threshold encoding was proposed in [19,29] to transfer n bits resulting in an inflated ciphertext of size $2n \log_2 q$. Efficiency is further reduced if only a part of the n bits is used, for example to transfer a 128-bit AES key. Moreover, the Ring-LWEEncrypt scheme suffers from random decryption errors so that redundancy in the message m is required to correct those errors. In the following we analyze a simple but effective way to reduce the ciphertext expansion without significantly affecting the error rate. This approach has been previously applied to homomorphic encryption schemes [9, Section 6.4], [10, Section 4.2] and the idea is basically to cut-off a certain number of least significant bits of c_2 since they mostly carry noise but only few information supporting the threshold decoding. We experimentally verified the applicability of this approach in practice with regard to concrete parameters by measuring the error rates for reduced versions of c_2 as shown in Table 1 ($u = 1$).

Table 1. Bit-error rate for the encryption and decryption of 160,000,000 bytes of plaintext when cutting off a certain number x of least significant bits of every coefficient of c_2 for the parameter set ($n = 256, q = 7681, s = 11.31$) where u is the parameter of the additive threshold encoding (see Algorithm 1) and $\pm\lceil 2s \rceil$ the tailcut bound. For a cutoff of 12 or 13 bits almost no message can be recovered.

u	**Cut-off x bits**	0	1	2	3	4	5	6	7	8	9	10	11
1	**Errors** (10^3)	46	46	45.5	45.6	46	46.5	48.6	56.1	94.4	381	5359	135771
	Error rate (10^{-5})	3.59	3.59	3.56	3.57	3.59	3.63	3.80	4.38	7.38	29.81	418.7	10610
2	**Errors**	26	20	26	27	23	21	21	32	71	957	125796	$44 \cdot 10^6$
	Error rate (10^{-8})	2.03	1.56	2.03	2.11	1.80	1.64	1.64	2.5	5.55	74.7	9830	$34 \cdot 10^5$

As it turns out the error rate does not significantly increase – even if we remove 7 least significant bits of every coefficient and thus have halved the size of c_2. It would also be possible to cut-off very few bits (e.g., 1 to 3) of c_1 at the cost of an higher error rate. A further extreme option to reduce ciphertext expansion is to omit whole coefficients of c_2 in case they are not used to transfer message bits (e.g., to securely transport a symmetric key). Note that this approach does not affect the concrete security level of the scheme as the modification does not involve any knowledge of the secret key or message and thus does not leak any further information. When compared with much more complicated and hardware consuming methods, e.g., the compression function for the Lyubashevsky signature scheme presented in [20], this straightforward approach is much more practical.

Decreasing the Error Rate. As noted above decryption of Ring-LWEEncrypt is prone to undesired message bit-flips with some small probability. Such a faulty decryption is certainly highly undesirable and can also negatively affect security properties. One solution can be the subsequent application of forward error correcting codes but such methods obviously introduce additional complexity in hardware or software. As another approach, the error probability can be lowered by modifying the threshold encoding scheme, i.e., instead of encoding one bit into each coefficient of c_2, a plaintext bit is now encoded into u coefficients of c_2. This additive threshold encoding algorithm is shown in Figure 1 where `encode` takes as input a plaintext bit-vector m of length $\lfloor \frac{n}{u} \rfloor$ and outputs the threshold encoded vector $\bar{m}$ of size m. The decoding algorithm is given the encoded message vector $\tilde{m}$ affected by an unknown error vector. The impact on the error rate by using additive threshold encoding ($u = 2$) jointly with the removal of least significant bits is shown in Table 1. Note that this significantly lowers the error rate without any expensive encoding or decoding operations and is much more efficient than, e.g., a simple repetition code [35].

Algorithm Encode($m = \{0,1\}^{\lfloor \frac{n}{u} \rfloor}, u$)

```
1: for i=0 to $\lfloor \frac{n}{u} \rfloor - 1$ do
2:     for j=0 to u-1 do
3:         $\bar{m}[u \cdot i + j] = m[i] \cdot \frac{q-1}{2}$
4:     end for
5: end for
6: return $\bar{m}$
```

Algorithm Decode($\tilde{m} = \{-\frac{q-1}{2}, \frac{q-1}{2}\}^n, u$)

```
1: for i=0 to $\lfloor \frac{n}{u} \rfloor$ do
2:     $s = 0$
3:     for j=0 to u-1 do
4:         $s = s + |\tilde{m}[u \cdot i + j]|$
5:     end for
6:     if $s < \frac{u \cdot q}{4}$ then
7:         $m[i] = 0$
8:     else
9:         $m[i] = 1$
10:    end if
11: end for
12: return $m$
```

Fig. 1. Additive threshold encoding.

3 Implementation of Ring-LWEEncrypt

In this section we describe the design and implementation of our processor with special focus on the efficient and flexible implementation of Gaussian sampling.

3.1 Gaussian Sampling

Beside its versatile applicability in lattice-based cryptography, sampling of Gaussian distributed numbers is also crucial in electrical engineering and information technology, e.g., for the simulation of complex communication systems (see [49] for a survey from this perspective). However, it is not clear how to adapt continuous Gaussian samplers, like the ones presented in [22,28,53], for the requirements of lattice-based cryptography. In the context of discrete Gaussian sampling for lattice-based cryptography the most straightforward method is rejection sampling. In this case an uniform integer $x \in \{-\tau\sigma, ..., \tau\sigma\}$, where τ is the "tail-cut" factor, is chosen from a certain range depending on the security parameter and then accepted with probability proportional to $e^{-x^2/2\sigma^2}$ [18]. This method has been implemented in software in [19] but the success rate is only approximately 20% and requires costly floating point arithmetic (cf. to the laziness approach in [14]). Another method is a table-based approach where a memory array is filled with Gaussian distributed values and selected by a randomly generated address. Unfortunately, a large resolution – resulting in a very large table – is required for accurate sampling. It is not explicitly addressed in [19] how larger values such as $x = \lceil 2s \rceil$ for $s = 6.67$ with a probability of $\Pr[x = 14] = 1.46 \cdot 10^{-7}$ are accurately sampled from a table with a total resolution of only 1024 entries. We further refer to [13, Table 2] for a comparison of different methods to sample from a Gaussian distribution and a new approach.

Hardware Implementation Using the Inverse Transform Method. Since the aforementioned methods seem to be unsuitable for an efficient hardware implementation we decided to use the inverse transform method. When applying this method in general a table of cumulative probabilities $p_z = \Pr(x \leqslant z : x \leftarrow D_\sigma)$ for integers $z \in [-\tau\sigma, ..., \tau\sigma]$ is computed with a precision of λ bits. For a uniformly random chosen value x from the interval $[0, 1)$ the integer $y \in \mathbb{Z}$ is then returned (still requiring costly floating point arithmetic) for which it holds that $p_{z-1} \leq x < p_z$ [13,16,41].

In hardware we operate with integers instead of floats by feeding a uniformly random value into a parallel array of comparators. Each comparator c_i compares its input to the commutative distribution function scaled to the range of the PRNG outputting r bits. As we have to cut the tail at a certain point, we compute the accumulated probability over the positive half (as it is slightly smaller than 0.5) until we reach the maximum value j (e.g., $j = \lceil 2s \rceil$) so that $w = \sum_{k=0}^{j} \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$. We then compute the values fed into the comparators as $v_k = \frac{2^{r-1}-1}{w}(v_{k-1} + \sum_{k=0}^{j} \rho_\sigma(x)/\rho_\sigma(\mathbb{Z}))$ for $0 < k \leq j$ and with $v_0 = \frac{2^{r-1}-1}{2w}\rho_\sigma(0)/\rho_\sigma(\mathbb{Z})$. Each comparator c_i is preloaded with the rounded value v_i and outputs a one bit if the input was smaller or equal to v_i. A subsequent circuit then identifies the first comparator c_l which returned a one bit and outputs either l or $-l$.

The block diagram of the sampler is shown in Figure 2 for the concrete parameter set $(n = 256, q = 7681, s = 11.32)$ where the output of the sampler is bound to $[-\lceil 2s \rceil, \lceil 2s \rceil] = [-5.09\sigma, 5.09\sigma]$ and the amount of required randomness is 25 bits per sample. These random bits are supplied by a PRNG for which we used the output of an AES block cipher operating in counter mode. Each 128-bit output block of our AES-based PRNG allows sampling of 5 coefficients. One random bit is used for sign determination while the other 24 bits form a uniformly random value. Finally, the output of the sampler is buffered in a FIFO. When leaving the FIFO, the values are lifted to the target domain

$[0, q-1]$. Although it is possible to generate a sampler directly in VHDL by computing the cumulative distribution function on-the-fly during synthesis, we have implemented a Python script for this purpose. The reason is that the VHDL floating point implementation only provides double accuracy while the `Decimal`[5] data type supports arbitrary precision. The Python script also performs a direct evaluation of the properties of the sampler (e.g., statistical distance).

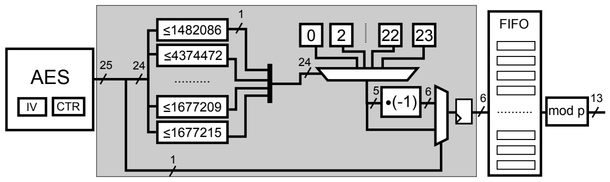

Fig. 2. Gaussian sampler using the inverse transform sampling method.

3.2 Ring-LWE Processor Architecture

The core of our processor is built around an NTT-based polynomial multiplier which is described in [43]. The freely available implementation has been further optimized and the architecture has been extended from a simple polynomial multiplier into a full-blown and highly configurable micro-code engine. Note that Aysu et al. [6] recently proposed some improvements to the architecture of [43] in order to increase the efficiency and area usage of the polynomial multiplier. While some improvements rely on their decision to fix the modulus q to $2^{16}+1$ other ideas are clearly applicable in future work and revisions of our implementations. However, we do not fix q as the design goal of our hardware processor is the native support for a large variety of ideal lattice-based schemes, including the most common operations on polynomials like addition, subtraction, multiplication by the NTT as well as sampling of Gaussian distributed polynomials. By supporting an arbitrary number of internal registers (each can store one polynomial) realized in block RAMs and by reusing the data path of the NTT multiplier for other arithmetic operations we achieve high performance at low resource consumption.

General Description and Instruction Set. The datapath of our engine depicted in Figure 3 depends on the size of the reduction prime q and is thus $\log_2 q$ as polynomial coefficients are processed serially in a pipeline. Four registers are fixed where register R0 and R1 are part of the NTT block, while the Gaussian sampler is connected to register R2. Register R3 is exported to upper layers and operates as I/O port. More registers R4 to Rx can be flexibly enabled during synthesis where each additional register can hold a polynomial with n elements of size $\log_2 q$. The `Switch matrix` is a dynamic multiplexer that connects registers to

[5] `http://docs.python.org/2/library/decimal.html`

the `ALU` and the external interface and is designed to process statements in two-operand form like $R1 \leftarrow R1+R2$. All additional registers Rx for $x > 4$ are placed inside of the `Register array` component. The `Decoder` unit is responsible for interpreting instructions that configure the switch matrix, determines whether the `ALU` has to be used (SUB, ADD, MOV) or if NTT specific commands need to invoke the `NTT multiplier`. To improve resource utilization of the overall system, the butterfly unit of the NTT core is shared between the NTT multiplier and the ALU.

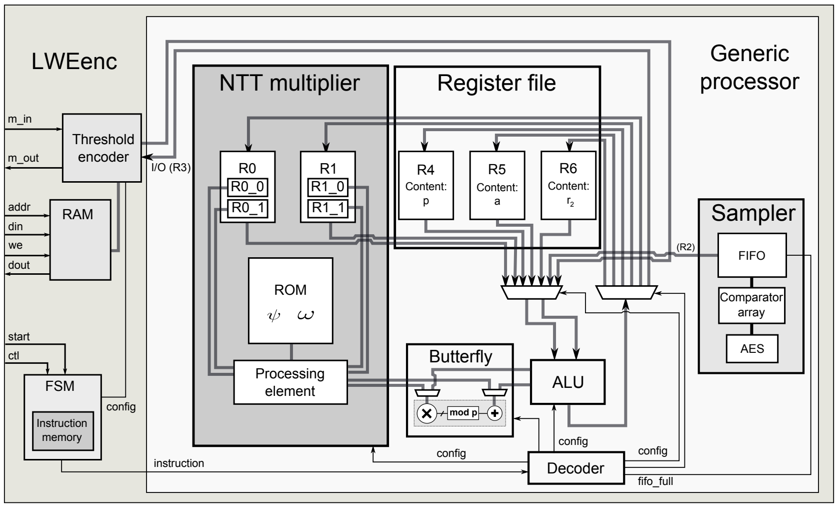

Fig. 3. Architecture of our implementation of the Ring-LWEEncrypt engine with a particular instance of our generic lattice processor with three additional registers R4-6.

The most important instructions supported by the processor are the iterative forward (NTT_NTT) as well as the backward transform (NTT_INTT) which take $\approx \frac{n}{2}\log_2 n$ cycles. Other instructions are for example used for the bit-reversal step (NTT_REV), point-wise multiplication (NTT_PW_MUL), addition (ADD), or subtraction (SUB) – each consuming $\approx n$ cycles. Note that the sampler and the I/O port are just treated as general purpose registers. Thus no specific I/O or sampling instructions are necessary and for example the MOV command can be used. Note also that the implementation of the NTT is performed in place and commands for the backward transformation (e.g., NTT_PW_MUL, or NTT_INTT) modify only register R1. Therefore, after a backward transform a value in R0 is still available.

Implementation of Ring-LWEEncrypt. For our implementation we used the medium and high security parameter sets as proposed in [19] which are specifically optimized for hardware. We further exploit the general characteristic of the

NTT which allows it to "decompose" a multiplication into two forward transforms and one backward transform. If one coefficient is fixed or needed twice it is wise to directly store it in NTT representation to save subsequent transformations. In Figure 4 the modified algorithm is given which is more efficient since the public constant a as well as the public and private keys p and r_2 are stored in NTT representation.

As a consequence, an encryption operation consists of a certain overhead, one forward NTT transformation ($n + \frac{1}{2}n \log_2 n$ cycles), two backward transforms ($2 \cdot (2n + \frac{1}{2}n \log_2 n)$ cycles), two coefficient-wise multiplications ($2n$ cycles), three calls to the Gaussian sampling routine ($3n$ cycles) and some additions as well as data movement operations ($3n$ cycles) which return the error vectors. For decryption, we just need two NTT transformations, one coefficient-wise multiplications and one addition.

Domain Parameters
Temporary value: $r_1 = \texttt{sample}()$, Global constant: $\tilde{a} = \texttt{NTT}(a)$
Secret key: $\tilde{r}_2 = \texttt{NTT}(\texttt{sample}())$, Public key: $\tilde{p} = \texttt{NTT}(r_1 - \texttt{INTT}(\tilde{a} \circ \tilde{r_2}))$

```
Algorithm Enc($\tilde{a}, \tilde{p}, m \in \{0,1\}^n$)
1: $e_1, e_2, e_3 = \texttt{sample}()$
2: $\tilde{e}_1 = \texttt{NTT}(e_1)$
3: $\tilde{h_1} = \tilde{a} \circ \tilde{e}_1, \tilde{h_2} = \tilde{p} \circ \tilde{e}_1$
4: $h_1 = \texttt{INTT}(\tilde{h}_1), h_2 = \texttt{INTT}(\tilde{h}_2)$
5: $c_1 = h_1 + e_2$
6: $c_2 = h_2 + e_3 + \texttt{encode}(m)$
```

```
Algorithm Dec($c_1, c_2, \tilde{r}_2$)
1: $\tilde{h}_1 = \texttt{NTT}(c_1)$
2: $\tilde{h}_2 = \tilde{c}_1 \circ \tilde{r}_2$
3: $m = \texttt{decode}(\texttt{INTT}(\tilde{h}_2) + c_2)$
```

Fig. 4. NTT-aware algorithms for Ring-LWEEncrypt.

The top-level module (`LWEenc`) in Figure 3 instantiates the ideal lattice processor and uses a block RAM as external interface to export or import ciphertexts c_1, c_2, keys r_2, p or messages m with straightforward clock domain separation (see again Figure 3). The processor is controlled by a finite state machine (`FSM`) issuing commands to the lattice processor to perform encryption, decryption, key import or key generation. It is configured with three general purpose registers R4-R6 in order to permanently store the public key p, the global constant a and the private key r_2. More registers for several key-pairs are also supported but optional. The implementation supports pre-initialization of registers so that all constant values and keys can be directly included in the (encrypted) bitstream. Note that, for encryption, the core is run similar to a stream cipher as c_1 and c_2 can be computed independently from the message which is then only added in the last step (e.g., comparable to the XOR operation used within stream ciphers).

4 Results and Performance

For performance analysis we primarily focus on Virtex-6 platforms (speed grade -2) but would also like to emphasize that our solution can be efficiently implemented even on a small and low-cost Spartan-6 FPGA. All results were obtained after post-place and route (Post-PAR) with Xilinx ISE 14.2.

4.1 Gaussian Sampling

In Table 2 we summarize resource requirements of six setups of the implemented comparator-based Gaussian sampler for different tail cuts and statistical distances. Our random number generator is a round based AES in counter mode that computes a 128-bit AES block in 13 cycles and comprises 349 slices, 1181/350 LUT/FF, two 18K block RAMs and runs with a maximum frequency of about 265 MHz. Combined with this PRNG[6], Gaussian sampling based on the inverse transform method is efficient for small values of s (as typically used for Ring-LWEEncrypt) but would not be suitable for larger Gaussian parameters like, e.g., $s = \sqrt{2\pi}2688 = 6737.8$ for the treeless signature scheme presented in [31]. While our sampler needs a huge number of random inputs, the AES engine is still able to generate these numbers (for each encryption we need $3n$ samples). Table 2 also shows that it is possible to realize an efficient sampler even for a small statistical distance $< 2^{-80}$ since its resource consumption of roughly 250 slices is quite moderate (setup III/IV). With additional register levels and pipelining for versions I/II we achieved the overall clock frequency for the whole core reported in Table 3 in this section. As the PRNG does not provide enough randomness to sample a value in every clock cycle it is not required to evaluate the comparator array in every single cycle so that in particular setups III-VI can use several clock cycles until output is provided. This lowers the critical path and thus allows higher clock frequencies without costs for pipelining registers. Setups V/VI are even more accurate and support (theoretical) requirements of a statistical distance smaller than 2^{-100} [16]. However, then a faster PRNG would be required as for $n = 256$ we would need $105 \cdot 3n = 80640$ bits of random input.

4.2 Performance of Ring-LWEEncrypt

Table 3 lists the resource consumption and performance of our implementation of Ring-LWEEncrypt. As stated in Section 3.2 our implementation combines key generation, encryption and decryption in a holistic design and would not significantly benefit from removing any one of these functional units. The only exception might be a decryption-only core in which no Gaussian sampling is needed.

Table 4 compares the results achieved in this work with the implementation by Göttert et al. [19] as well as other relevant asymmetric schemes and also

[6] Generation of true random numbers is not in the scope of this work; we refer to the survey by Varchola [50] how to achieve this.

Table 2. Performance, resource consumption, and quality of the core part (shaded grey in Figure 2) of the Gaussian sampler on a Virtex-6 LX75T (Post-PAR). The entry *rnd* denotes the number of used random bits to sample one value.

Setup	s	Max s	*rnd*	Slices	LUT/FF	MHz	Stat. Distance
I	11.32	23	25	42	136/5	115	$< 2^{-22}$
II	12.18	25	25	46	149/5	118	$< 2^{-22}$
III	11.32	48	85	231	863/6	61	$< 2^{-80}$
IV	12.18	51	85	255	911/6	61	$< 2^{-80}$
V	11.32	53	105	314	1157/6	58	$< 2^{-100}$
VI	12.18	57	105	342	1248/6	50	$< 2^{-100}$

Table 3. Resource consumption and performance of the combined key generation, encryption and decryption engine for the two different security levels on a Virtex-6 LX75T (Post-PAR). The public key requires $n \log_2 q$ bits (when stored in NTT representation), the private key $n \log_2 q$ bits and the ciphertext $2n \log_2 q$ bits.

	Aspect	Medium Security (n=256,q=7681,s=11.32)	High Security (n=512,q=12289,s=12.18)
Resources	Slices	1506	1887
	LUT/FF	4549/3624	5595/4760
	18K BRAM	12	14
	DSP48E1	1	1
Performance	MHz	262	251
	Key generation (cycles/time)	7235/27.61 µs	14532/57.90 µs
	Encryption (cycles/time)	6861/26.19 µs	13769/54.86 µs
	Decryption (cycles/time)	4404/16.80 µs	8883/35.39 µs

adds performance figures for a Spartan-6 instantiation. Note that a detailed comparison with [19] is unfair due to inaccuracies of synthesis results (the Virtex-6 LX240T FPGA used in [19] was overmapped so that the subsequent place-and-route (PAR) step providing final results could not be performed). Figures for clock frequency, overall slice consumption, and cycles counts for individual operations or the whole encryption block are thus not given in [19]. We therefore can only refer to numbers providing the resource consumption of registers and LUT usage. For a rough comparison we apply the throughput to area (T/A) metric and define area equivalent to the usage of LUTs due to the restriction mentioned above. It turns out that our implementation for $n = 256$ is 32 times smaller regarding key generation, 65 times smaller for encryption and 27 times smaller for decryption, at a loss of a factor of about 2 and 3.3 in performance. When employing the $\frac{\text{Bit/s}}{\text{LUT}}$ metric for medium security encryption we achieve $\frac{9.77 \cdot 10^6 \text{Bits}}{4549 \text{ LUTs}} = 2147$ while the work presented in [19] gives $\frac{31.8 \cdot 10^6 \text{Bits}}{298016 \text{ LUTs}} = 106$. This results in an improvement of a factor of roughly 20.[7]

[7] For this comparison we assumed that for each encryption 256 bits are transmitted.

In comparison with a recent implementations of the code-based Niederreiter scheme [24] we are faster for decryption and we also use fewer resources on the same platform. Another natural target for comparison is the patent-protected NTRU scheme which has been implemented on a large number of architectures [5, 7, 23]. The implementation in [27] is clearly faster than ours. However, the implemented NTRU(251,3,12) variant in [27] seems to be less secure than our scheme [25]. Unfortunately, we are not aware of any newer NTRU FPGA implementations in order to determine the impact of increased security parameters on runtime and area consumption. In software, NTRU even seems to be rather slow for higher security levels what can be obtained from the 256-bit secure NTRU software implementation (`ntruees787ep1`) benchmarked using the eBACS framework [8] with secret/public key sizes of 1854/1574 bytes and a ciphertext of 1574 bytes. For the ideal lattice-based NTRU version presented in [47], no implementation and concrete parameters have been published yet. In comparison with ECC over prime curves (i.e., a single point multiplication [21]) and RSA (random-exponent 1024-bit exponentiation [48]) our implementation is by an order of magnitude faster, scales better for higher security levels, and also consumes less resources. However, we are not able to beat the recent binary curve implementation of Rebeiro et al. [45] in terms of throughput and performance.

4.3 Constant Time Operation

Side-channel attacks are a problem for all physical implementations [36]. A simple target for a side-channel attack is the use of timing information of the security algorithm by measuring execution time or cycles. Our implementation of Ring-LWEEncrypt is fully pipelined and has no data-dependent operations. The processor core does not support any branches and Gaussian sampling based on the inverse transform operates in constant time. Summarizing, all cryptographic operations of our core are timing-invariant.

5 Conclusions and Future Work

In this work we presented a novel implementation of the ideal lattice-based Ring-LWE encryption scheme that fits even on a low-cost Spartan-6 FPGA. According to our findings, we improved the results obtained in the previous work of [19] by at least an order of magnitude using the same FPGA platform and much less resources.

Future work can combine our hardware engine with error correction facilities and CCA2 conversion. Additionally, countermeasures against further side-channel and fault-injection attacks need to be considered. As we intend to make our implementation publicly available, our work also offers the chance for third-party side-channel evaluation and cryptanalysis (e.g., exploiting the concrete implementation of the Gaussian sampler). Since our processor could also be utilized by other lattice-based cryptosystems, the provably secure NTRU variant presented in [47] can be another target for implementation. Moreover, a recent

Table 4. Performance comparison of our proposal with other public key encryption schemes ($\approx$ 80..128 bit) comparable to the medium security ($n = 256, q = 7681, s = 11.31$) parameter set which is capable of transferring 256-bit messages. Our implementation is versatile enough to perform encryption, decryption and key generation in a single core. Figures denoted with an asterisk (*) are less accurate results obtained from synthesis due to extensive overmapping of resources.

Scheme	**Device**	**Resources**	**Speed**
Our Work [Gen/Enc/Dec] (n=256)	S6LX16 @160 MHz	4121 LUT/3513 FF/ 14 BRAM(8K)/1 DSP48	45.22 µs 42.88 µs 27.51 µs
Our Work [Gen/Enc/Dec] (n=256)	V6LX75T @262 MHz	4549 LUT/3624 FF/ 12 BRAM(18K)/1 DSP48	27.61 µs 26.19 µs 16.80 µs
Ring-LWEEncrypt [Gen/Enc/Dec] (n=256) [19]	V6LX240T V6LX240T V6LX240T	146718 LUT/82463 FF 298016 LUT/143396 FF 124158 LUT/65174 FF	- 8.05 µs* 8.10 µs
Niederreiter [Enc/Dec] [24]	V6LX240T V6LX240T	888 LUT/875 FF/17 BRAM 9409 LUT/12861 FF/ 12 BRAM	0.66 µs 57.78 µs
NTRU [Enc/Dec] [27]	XCV1600E	27292 LUT/5160 FF	1.54 µs 1.41 µs
1024-bit mod. Exp. [48]	XC4VFX12	3937 SLICE/17 DSP48	1.71 ms
ECC-P224 [21]	XC4VFX12	1825 LUT/1892 FF/ 26 DSP48/ 11 BRAM	365.1 µs
ECC-B233 [45]	XC5VLX85T	18097 LUT/5644 SLICE	12.3 µs

proposal of a lattice-based signature scheme by Ducas et al. [13] uses exactly the same parameters ($n = 512, q = 12289$) as Ring-LWEEncrypt and is thus a natural target for implementation based on our micro-code engine.

References

1. Shweta Agrawal, Dan Boneh, and Xavier Boyen. Efficient lattice (H)IBE in the standard model. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 553–572. Springer, 2010.
2. M. Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108. ACM, 1996.
3. Martin Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the Complexity of BKW Algorithm against LWE. In *SCC'12: Proceedings of the 3nd International Conference on Symbolic Computation and Cryptography*, pages 100–107, Castro-Urdiales, July 2012.

4. Martin Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the Complexity of the Arora-Ge algorithm against LWE. In *SCC '12: Proceedings of the 3nd International Conference on Symbolic Computation and Cryptography*, pages 93–99, Castro-Urdiales, July 2012.
5. Ali Can Atici, Lejla Batina, Junfeng Fan, Ingrid Verbauwhede, and Siddika Berna Örs. Low-cost implementations of NTRU for pervasive security. In *ASAP*, pages 79–84. IEEE Computer Society, 2008.
6. Aydin Aysu, Cameron Patterson, and Patrick Schaumont. Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 2013*. IEEE, 2013. to appear.
7. Daniel V. Bailey, Daniel Coffin, Adam J. Elbirt, Joseph H. Silverman, and Adam D. Woodbury. NTRU in constrained devices. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 2001.
8. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. `http://bench.cr.yp.to` (accessed 2013-05-10).
9. Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2013:75, 2013.
10. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.
11. Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *STOC*, pages 575–584. ACM, 2013.
12. J. Buchmann, A. May, and U. Vollmer. Perspectives for cryptographic long-term security. *Communications of the ACM*, 49(9):50–55, 2006.
13. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. *IACR Cryptology ePrint Archive*, 2013:383, 2013. To appear at CRYPTO 2013.
14. Léo Ducas and Phong Q. Nguyen. Faster Gaussian lattice sampling using lazy floating-point arithmetic. In Wang and Sako [51], pages 415–432.
15. Léo Ducas and Phong Q. Nguyen. Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures. In Wang and Sako [51], pages 433–450.
16. S.D. Galbraith and N.C. Dwarakanath. Efficient sampling from discrete gaussians for lattice-based cryptography on a constrained device.
17. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178. ACM, 2009.
18. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Cynthia Dwork, editor, *STOC*, pages 197–206. ACM, 2008.
19. Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Prouff and Schaumont [44], pages 512–529.
20. Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Prouff and Schaumont [44], pages 530–547.

21. Tim Güneysu and Christof Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2008.
22. Roberto Gutierrez, V. Torres, and Javier Valls. Hardware architecture of a Gaussian noise generator based on the inversion method. *IEEE Trans. on Circuits and Systems*, 59-II(8):501–505, 2012.
23. Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In Josef Pieprzyk, editor, *CT-RSA*, volume 5985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2010.
24. Stefan Heyse and Tim Güneysu. Towards one cycle per bit asymmetric encryption: Code-based cryptography on reconfigurable hardware. In Prouff and Schaumont [44], pages 340–355.
25. Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 437–455, 2009.
26. J. Hoffstein, J. Pipher, and J. Silverman. NTRU: A ring-based public key cryptosystem. *Algorithmic number theory*, pages 267–288, 1998.
27. A.A. Kamal and A.M. Youssef. An FPGA implementation of the NTRUEncrypt cryptosystem. In *Microelectronics (ICM), 2009 International Conference on*, pages 209–212. IEEE, 2009.
28. D.-U. Lee, W. Luk, J.D. Villasenor, Guanglie Zhang, and P.H.-W. Leong. A hardware Gaussian noise generator using the Wallace method. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(8):911–920, 2005.
29. Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.
30. V. Lyubashevsky and D. Micciancio. Generalized compact knapsacks are collision resistant. *Automata, Languages and Programming*, pages 144–155, 2006.
31. Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.
32. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2008.
33. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, pages 1–23, 2010.
34. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *IACR Cryptology ePrint Archive*, 2012:230, 2012.
35. Florence MacWilliams and Neil Sloane. The theory of error-correcting codes. 2006.
36. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007 edition, 3 2007.
37. D. Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007.
38. Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. *IACR Cryptology ePrint Archive*, 2013:69, 2013.
39. Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud com-*

puting security workshop, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

40. Phong Q. Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 271–288. Springer, 2006.
41. Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2010.
42. J. M. Pollard. The fast fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.
43. Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2012.
44. Emmanuel Prouff and Patrick Schaumont, editors. *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*. Springer, 2012.
45. Chester Rebeiro, Sujoy Sinha Roy, and Debdeep Mukhopadhyay. Pushing the limits of high-speed GF(2^m) elliptic curve scalar multiplication on FPGAs. In Prouff and Schaumont [44], pages 494–511.
46. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *STOC*, pages 84–93. ACM, 2005.
47. Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2011.
48. D. Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 272–288, 2007.
49. David B. Thomas, Wayne Luk, Philip Heng Wai Leong, and John D. Villasenor. Gaussian random number generators. *ACM Comput. Surv.*, 39(4), 2007.
50. Michal Varchola. *FPGA Based True Random Number Generators for Embedded Cryptographic Applications*. PhD thesis, Technical University of Kosice, 2008.
51. Xiaoyun Wang and Kazue Sako, editors. *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*. Springer, 2012.
52. Franz Winkler. *Polynomial Algorithms in Computer Algebra (Texts and Monographs in Symbolic Computation)*. Springer, 1 edition, 8 1996.
53. Guanglie Zhang, P.H.-W. Leong, Dong-U Lee, J.D. Villasenor, R. C C Cheung, and W. Luk. Ziggurat-based hardware Gaussian random number generator. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 275–280, 2005.

A Appendix

A.1 Using the NTT for Polynomial Multiplication

A method achieving complexity of $\mathcal{O}(n \log n)$ for polynomial multiplication is the Number Theoretic Transform (NTT) [42]. For a given primitive n-th root of unity

ω the generic forward $\mathrm{NTT}_\omega(a)$ of a sequence $\{a_0, .., a_{n-1}\}$ to $\{A_0, \ldots, A_{n-1}\}$ with elements in $\mathbb{Z}_q$ and length n is defined as $A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \bmod q, \; i = 0, 1, ..., n-1$ with the inverse $\mathrm{NTT}_\omega^{-1}(A)$ just using ω^{-1} instead of ω.

The NTT can also be used directly for multiplication of polynomials in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ when ω is a primitive n-th root of unity in $\mathbb{Z}_q$ and $\psi^2 = \omega$ which is guaranteed by the requirement that $p = 1 \mod 2n$. When $a = (a_0, ..., a_{n-1})$ and $b = (b_0, ..., b_{n-1})$ be vectors of length n with elements in $\mathbb{Z}_q$ let $d = (d_0, ..., d_{n-1})$ be the negative wrapped convolution of a and b (thus $d = a * b \mod x^n + 1$). Let $\bar{a}, \bar{b}$ and $\bar{d}$ be defined as $(a_0, \psi a_1, ..., \psi^{n-1} a_{n-1})$, $(b_0, \psi b_1, ..., \psi^{n-1} b_{n-1})$, and $(d_0, \psi d_1, ..., \psi^{n-1} d_{n-1})$. Then $\bar{d} = NTT_w^{-1}\ (NTT_w(\bar{a}) \circ NTT_w(\bar{b}))$ [52].

The implementation given in [43] uses the standard Cooley and Tukey radix-2 decimation in time approach to compute the NTT. The core operation is the multiplication of the factor $\omega^{N \bmod n}$ with d and the adding or subtraction of the result from c (the famous "butterfly"). By *Bit-Reverse(g)* the input vector g is reordered where the new position of an element at position k is determined by the value obtained by reversing the binary representation of k.

A.2 Detailed Processor Instructions and Implementation

In Table 5 we provide a list of supported micro-code instructions for our ideal lattice processor and show in Table 6 how we used these instructions to implement the Ring-LWEEncrypt encryption scheme defined in Figure 4.

Table 5. The basic instruction set of the ideal lattice processor. Note that between every instruction a certain number of wait cycles ϵ (approx. 40 depending on pipeline size) is necessary in order to clear the pipeline and reconfigure the switch matrix.

Command	**Op 1**	**Op 2**	**Cycles**	**Explanation**
NTT_REV_{A/B}	$R\{2..x\}$	-	$n+\epsilon$	Loads a polynomial into register R{0/1} of the NTT engine, performs the bit reversal step and multiplies with NTT constants.
NTT_NTT_{A/B}	-	-	$\frac{n}{2}\log n+\epsilon$	Executes the NTT on register R{0/1}.
NTT_PW_MUL	-	-	$n+\epsilon$	Point/Coefficient-wise multiplication of registers R0 and R1. The result is stored in register R1.
NTT_INTT	-	-	$\frac{n}{2}\log n+\epsilon$	Executes the inverse NTT on register R1.
NTT_INV_PSI	-	-	$n+\epsilon$	Multiplies coefficients in R1 and multiplies with NTT constants.
NTT_INV_N	-	-	$n+\epsilon$	Multiplies coefficients in R1 with n^{-1}.
ADD	$R\{0..x\}$	$R\{0..x\}$	$n+\epsilon$	Adds two polynomials ($R(op1) \leftarrow R(op1)+R(op2)$)
SUB	$R\{0..x\}$	$R\{0..x\}$	$n+\epsilon$	Subtracts two polynomials ($R(op1) \leftarrow R(op1)-R(op2)$)
MOV	$R\{0..x\}$	$R\{0..x\}$	$n+\epsilon$	Moves a polynomial from one to another register ($R(op1) \leftarrow R(op2)$).
WAIT_SAMPLER	-	-	ϵ	Waits until the sampler has buffered more than n coefficients.
NTT_GP_MODE	-	-	ϵ	Export special purpose NTT registers as general purpose registers until the next NTT operation.
EN_CPIO	-	-	ϵ	Enables a mode in which polynomials written to the target register are also written to the I/O port.
DIS_CPIO	-	-	ϵ	Disables the copy to I/O mode.

Table 6. Encryption and decryption program for the Ring-LWEEncrypt scheme executed by our implementation. The public key p is stored in R4, the global constant a in R5, and the private key r_2 in R6. The sampler port is R2 and the I/O port is R3.

Encryption	**Comment**	**Decryption**	**Comment**
NTT_GP_MODE		NTT_BITREV_B(3)	Load c_1 into R1
MOV(1, 5)	Load $\texttt{NTT}(a)$ into R1	NTT_GP_MODE	
WAIT_SAMPLER		MOV(0, 6)	Load $\texttt{NTT}(r_2)$ into R0
NTT_BITREV_A(2)	Sample/Load e_1 into NTT	NTT_NTT_B	Compute $\texttt{NTT}(c_1)$
NTT_NTT_A	R0 now contains $\texttt{NTT}(e_1)$	NTT_PW_MUL	
NTT_POINTWISE_MUL		NTT_INTT	
NTT_INTT		NTT_INV_PSI	
NTT_INV_PSI		NTT_INV_N	R1 now contains $r_2 \cdot c_1$
NTT_INV_N	R1 now contains $a \cdot e_1$	NTT_GP_MODE	
WAIT_SAMPLER		ADD(1, 3)	Add c_2 to $r_2 \cdot c_1$
NTT_GP_MODE		MOV(3, 1)	Output $c_1 \cdot r_2 + c_2$
EN_CPIO	Copy result to I/O		
ADD(1, 2)	Sample/add e_2 to $a \cdot e_1$		
DIS_CPIO			
MOV(1, 4)	Load $\texttt{NTT}(p)$ into R1		
NTT_PW_MUL	R0 still contains $\texttt{NTT}(e_1)$		
NTT_INTT			
NTT_INV_PSI			
NTT_INV_N	R1 now contains $p \cdot e_1$		
WAIT_SAMPLER			
NTT_GP_MODE			
EN_CPIO	Copy result to I/O		
ADD(1, 2)	Sample/add e_3 to $p \cdot e_1$		
DIS_CPIO			