



**HAL**  
open science

## Efficient Hardware/Software Co-design for NTRU

Tim Fritzmann, Thomas Schamberger, Christoph Frisch, Konstantin Braun,  
Georg Maringer, Martha Johanna Sepulveda Florez

► **To cite this version:**

Tim Fritzmann, Thomas Schamberger, Christoph Frisch, Konstantin Braun, Georg Maringer, et al.. Efficient Hardware/Software Co-design for NTRU. 26th IFIP/IEEE International Conference on Very Large Scale Integration - System on a Chip (VLSI-SoC), Oct 2018, Verona, Italy. pp.257-280, 10.1007/978-3-030-23425-6\_13 . hal-02321771

**HAL Id: hal-02321771**

**<https://inria.hal.science/hal-02321771v1>**

Submitted on 21 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.



Distributed under a Creative Commons Attribution 4.0 International License

# Efficient Hardware/Software Co-Design for NTRU

Tim Fritzmann, Thomas Schamberger, Christoph Frisch, Konstantin Braun,  
Georg Maringer, Johanna Sepúlveda

Technical University of Munich, Germany

{tim.fritzmann,t.schamberger,chris.frisch,konstantin.braun,georg.maringer,  
johanna.sepulveda}@tum.de

**Abstract.** The fast development of quantum computers represents a risk for secure communications. Current traditional public-key cryptography will not withstand attacks performed on quantum computers. In order to prepare for such a quantum threat, electronic systems must integrate efficient and secure post-quantum cryptography which is able to meet the different application requirements and to resist implementation attacks. The NTRU cryptosystem is one of the main candidates for practical implementations of post-quantum public-key cryptography. The standardized version of NTRU (IEEE-1363.1) provides security against a large range of attacks through a special padding scheme. So far, NTRU hardware and software solutions have been proposed. However, the hardware solutions do not include the padding scheme or they use optimized architectures that lead to a degradation of the security level. In addition, NTRU software implementations are flexible but most of the time present a low performance when compared to hardware solutions. In this work, for the first time, we present a hardware/software co-design approach compliant with the IEEE-1363.1 standard. Our solution takes advantage of the flexibility of the software NTRU implementation and the speedup due to the hardware accelerator specially designed in this work. Furthermore, we provide a refined security reduction analysis of an optimized NTRU hardware implementation presented in a previous work.

**Keywords:** Lattice-based Cryptography, NTRU, HW/SW co-design, Side-Channel Attack

## 1 Introduction

Public-key cryptography (PKC) provides the basis for establishing secured communication channels between multiple parties. It supports the confidentiality, authenticity and non-repudiation of electronic communications and data storage. Internet-of-Things (IoT) and Cloud computing are some of the technologies that use PKC to secure channels. Traditional PKC such as the Rivest-Shamir-Adleman (RSA) cryptosystem, which is based on the factorization of larger numbers, or Elliptic Curve Cryptography (ECC), which is based on the discrete loga-

rithm problem, are considered insecure to attacks performed by a quantum computer. The foreseeable breakthrough of quantum computers therefore represents a risk for all communication systems. By executing Shor’s [23] and Grover’s [17] quantum algorithms, these computers will be able to solve the problems, on which classical PKC (RSA, ECC) relies, in polynomial time and to decrease the security level of symmetric cryptosystems, respectively. While symmetric cryptosystems can be easily secured against quantum computers by choosing larger key sizes, securing PKC requires new hard mathematical problems. To ensure long-term communication security, quantum-resistant (also called post-quantum) cryptography must be adopted. Post-quantum cryptography relies on mathematical problems that are secure against attacks from both traditional and quantum computers.

The skyrocketing evolution of quantum computers in particular poses a significant threat for applications with long life-cycles (e.g., cars, airplanes and satellites), where deployed devices are hard to update. As a reaction, the National Institute of Standards and Technology (NIST) started the process for post-quantum standardization in 2017 [19]. The goal is to select a set of appropriate post-quantum solutions which are able to meet the security, performance, cost, and adaptability requirements of current and future applications. Lattice-based cryptography is among the most promising post-quantum solutions. The lattice-based cryptosystem NTRU is one of the main alternatives for practical implementations of post-quantum PKC. NTRU is characterized by small key sizes (low memory footprint) and computational efficiency when compared to other post-quantum approaches [6, 7, 22]. NTRU has been standardized in the IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices (IEEE-1363.1) [11].

Empowering electronic devices with strong security poses several challenges due to limited resources, strict performance requirements, a tight time-to-market window, and the vulnerability to implementation attacks. To cope with the ever-increasing complexity when designing an embedded system both hardware and software solutions are usually explored. Embedded system design is based on powerful design strategies, such as co-design, where the algorithm tasks are split and implemented through hardware and software elements, resulting in high-speed and flexible implementations. In addition, security solutions do not only rely on efficient implementations, but they should be resistant to attacks, such as Chosen-Ciphertext Attacks (CCA) or Side-Channel Attacks (SCA). Adversaries can recover the secret key by gathering information obtained through the decryption of fabricated ciphertexts or by the physical information leakage during the cryptographic operation due to, e.g., timing, power consumption, and electromagnetic radiation. CCA can be avoided by adopting a padding scheme such as the Short Vector Encryption Scheme (SVES) which is defined by the NTRU standard (IEEE-1361.1). SCA on the other hand requires a careful implementation of the cryptographic algorithm.

NTRU hardware implementations have been demonstrated in [1, 3, 13, 15, 16, 25]. While current hardware solutions focus on efficient convolution techniques,

a complete hardware implementation of the standardized NTRU is still missing. Moreover, security aspects of the implementation are still largely unexplored. The works presented in [1, 3, 13, 15, 16, 25] do not implement the SVES padding scheme and have no protection against CCA. Furthermore, the NTRU optimization presented in [16] reduces the security of NTRU by leaking information regarding the secret key as the execution time of this implementation depends on the value of the secret key. This makes an implementation impractical for real applications.

NTRU software implementations have been demonstrated in [2, 5, 14, 18]. Despite showing good results, all of the aforementioned works present at least one of the following drawbacks: i) NTRU parameters are already deprecated; ii) Protection against CCA is not considered; and iii) the implementations are based on outdated microprocessor/microcontroller architectures. Only the works of [6, 21, 24] present a complete NTRU software solution. The software solution proposed in [24] is the official NTRU implementation and [6, 21] are implementations tailor-made for ARM Cortex M0 and protected against timing as well as cache attacks.

While NTRU hardware implementations present better performance, software implementations are easier to develop and to maintain. The exploration of complete hardware solutions and co-design techniques are essential for practically implementing NTRU on embedded devices.

This work extends our previous contribution presented in [4], where we demonstrate the first complete, compact, and secure NTRU hardware implementation and show that the optimized NTRU implementation in [16] exhibits a timing side-channel by giving a bounded security reduction analysis. In this extended work, we present the first HW/SW co-design NTRU solution compliant with the IEEE 1363.1 standard. It takes advantage of the flexibility of the software implementation and the speedup through the design of a specific hardware accelerator. In addition, we present the refinement of our previous security reduction analysis and we state the exact result instead of a bound. In summary, the contributions of the paper are:

- First complete NTRU hardware implementation which includes the SVES padding scheme;
- A compact NTRU implementation able to execute encryption and decryption operations;
- HW/SW co-design of NTRU by outsourcing polynomial multiplication and modulo reduction to hardware;
- A security analysis of the previous NTRU implementation presented in [16] and demonstration of the exact security reduction;
- Performance and cost evaluation of our NTRU implementations.

The remainder of this article is organized as follows: Section 2 gives an overview of the previous works on NTRU hardware implementations. Section 3 describes the instantiation of NTRU with the SVES padding scheme. Section 4 and Section 5 present our complete NTRU hardware and HW/SW co-design implementations. Section 6 describes the security analysis of the optimized NTRU

presented in [16]. The experimental results are provided in Section 7. A conclusion is given in Section 8.

## 2 Related Works

The probably first NTRU encryption hardware implementation was proposed by Bailey *et al.* in 2001 [3]. To speed up the polynomial multiplication, which is usually the performance bottleneck of NTRU, the authors propose to scan the coefficients of the blinding polynomial  $r$ . For each non-zero coefficient, the public key  $h$  is added to a temporary result. Atıcı proposed the first encryption and decryption NTRU hardware implementation. The architecture includes power saving methods such as clock gating and partially rotating registers [1]. The implementation of Kamal *et al.* uses the special structure of the public key, which has a large number of zero coefficients to optimize the performance [13]. In [15], Liu *et al.* use the fact that the polynomial multiplication in the truncated polynomial ring of NTRU can be modeled with a Linear Feedback Shift Register (LFSR) to implement the polynomial multiplication. In [16], they speed up their implementation by skipping the multiplication operation when two consecutive zero coefficients in the ternary polynomial are detected. Thus, the multiplication time depends on the number of double zeros contained in the polynomial. This information decreases the NTRU security level as discussed in [4] and now refined in Section 6.

Moreover, so far none of the existing works has proposed a full hardware implementation of NTRU with the SVES padding scheme as defined in the IEEE-1363.1 standard. The work of [10] discusses different hardware design strategies for NTRU. However, it presents neither implementation results nor a detailed description of their approach that would allow to reproduce their architecture. Furthermore, the polynomial multiplication suggested in this work requires  $N^2$  operations whereas recent works only require  $N$  clock cycles. Hardware design strategies required for the SVES padding scheme were not discussed. As the integration of SVES is mandatory to inhibit CCA, implementations that only integrate polynomial multiplication in hardware show a misleading picture of the implementation costs. A commonly used tool for transforming cryptographic algorithms into CCA secured schemes is the NAEP transformation [9]. SVES is a concrete instantiation of the NAEP transform, which was specifically designed for NTRU. The first iterations, SVES-1 and SVES-2, are vulnerable to attacks exploiting decryption errors [8]. The latest iteration, SVES-3, which is sometimes only referred to as SVES, does not show this vulnerability. It is standardized in IEEE-1363.1 [11]. In contrast to previous works, we present a complete CCA-secure NTRU hardware implementation compliant with the standard.

Regarding the NTRU software implementations, many of them are not compliant with the standard, use outdated parameter sets or are tailored for a specific platform. The NTRUOpenSourceProject provides the official software reference implementation for NTRU that is fully compliant with the standard [24]. This implementation builds the basis for our HW/SW co-design. We accelerate the

performance critical multiplications during the encryption and decryption routines by outsourcing them to hardware. Our proposal combines the efficiency of a hardware design with the flexibility of a software design.

### 3 NTRU

#### 3.1 Notation

The main elements of NTRU are the polynomials in the following integer rings:

$$R_{N,p} = \frac{(\mathbb{Z}/p\mathbb{Z})[x]}{(x^N - 1)}, \quad R_{N,q} = \frac{(\mathbb{Z}/q\mathbb{Z})[x]}{(x^N - 1)}. \quad (1)$$

These rings define each polynomial to be at most of degree  $N - 1$  and to have integer coefficients. For  $R_{N,p}$  and  $R_{N,q}$  these coefficients are reduced modulo  $p$  and modulo  $q$ , respectively. Unless otherwise noted, all polynomials are elements of the ring  $R_{N,q}$ .

For the standardized parameter sets of NTRU the modulus  $p$  is fixed to a small prime  $p = 3$ . In this case, the elements of the ring  $R_{N,p}$  are called ternary polynomials. A ternary polynomial  $\mathcal{T}_N(d_1, d_2)$  has  $d_1$  coefficients equal to one and  $d_2$  coefficients equal to minus one, while the remaining coefficients are set to zero. Ternary polynomials in NTRU are sparse, that is, the majority of coefficients are set to zero. The values  $d_1$  and  $d_2$  are part of the parameter set and can be changed to achieve different security levels.

Additionally, the parameter  $q$  is fixed to  $q = 2048$ , which simplifies the implementation of the algorithm. By choosing the modulus to be a power of two, the modulo operation can be performed without additional costs by considering only the corresponding least significant bits.

#### 3.2 Short Vector Encryption Scheme (SVES)

In order to provide a CCA-secure encryption algorithm, NTRU is instantiated with the SVES. SVES defines a general padding scheme for the message which prevents CCA by identifying invalid ciphertexts using two auxiliary methods: i) Blinding Polynomial Generation Method (BPGM); and ii) Mask Generation Function (MGF). The general description of these two methods is presented in the next paragraphs. Implementation-specific details are given in Subsection 4.2 and Subsection 4.3.

**Blinding Polynomial Generation Method (BPGM).** This method generates an ephemeral blinding polynomial  $r$  in a deterministic way with the use of a Pseudo-Random Number Generator (PRNG). This PRNG is based on a hash function  $\mathbf{G}$  and is initialized by a seed consisting of four values:

$$r = \text{BPGM}(\text{OID}, m, b, h_{\text{trunc}}) . \quad (2)$$

The identifier  $OID$  is a unique three-byte value which depends on the parameter set. The parameter  $b$  is a random number and  $m$  the message to be encrypted. Finally,  $h_{trunc}$  is a pre-defined part of the public key in binary representation.

**Mask Generation Function (MGF).** Similar to the BPGM module, the MGF module uses a hash function  $\mathbf{G}$  to generate a mask. The input of  $\mathbf{G}$  is the result of the polynomial multiplication of the ephemeral blinding polynomial  $r$  and the public key  $h$ :

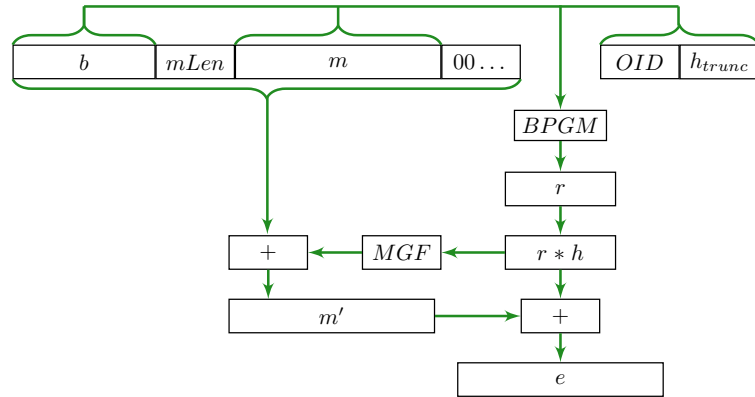
$$m_{mask} = MFG(r * h) . \quad (3)$$

The resulting mask  $m_{mask}$  is added to the message  $m$ .

### 3.3 NTRU with SVES

The NTRU scheme instantiated with SVES consists of the three cryptographic operations: key generation, encryption and decryption.

The key generation step creates a key pair, consisting of the public key  $h$  with its corresponding secret key  $f$ , through three steps. The first step generates two random ternary polynomials,  $F \in R_{N,p}$  and  $g \in R_{N,p}$ . The positions of the polynomial coefficients with value one and minus one are selected based on a uniform distribution. The second step calculates the private key  $f$  as  $f = 1 + pF$  together with its inverse  $f^{-1}$  modulo  $q$ . Not all the polynomials have an inverse in the corresponding ring. Therefore, it is possible that the inverse  $f^{-1}$  cannot be found. In this case, the key generation is restarted until a key with a valid inverse is found. The third step computes the public key  $h$  as  $h = pf^{-1} * g$ .



**Fig. 1.** NTRU encryption with SVES

The NTRU encryption is shown in Fig. 1. It transforms a message  $m$  into a ciphertext  $e$  through four steps. The first step encodes  $m$  into a ternary polynomial representation and concatenates this polynomial together with the random

number  $b$ , the identifier  $OID$ , and  $h_{trunc}$  as input to the BPGM. Then the ephemeral polynomial  $r$  is created as  $r = BPGM(OID || m || b || h_{trunc})$ . The second step multiplies  $r$  with the public key polynomial  $h$ . This result is used as an input to the MGF in order to obtain a mask as  $m_{mask} = MGF(r * h)$ . The third step adds the mask to the padded message  $m_{pad} = (b || mLen || m || 00\dots)$  to produce  $m' = m_{pad} + m_{mask}$ . The final step computes the ciphertext as  $e = m' + r * h$ .

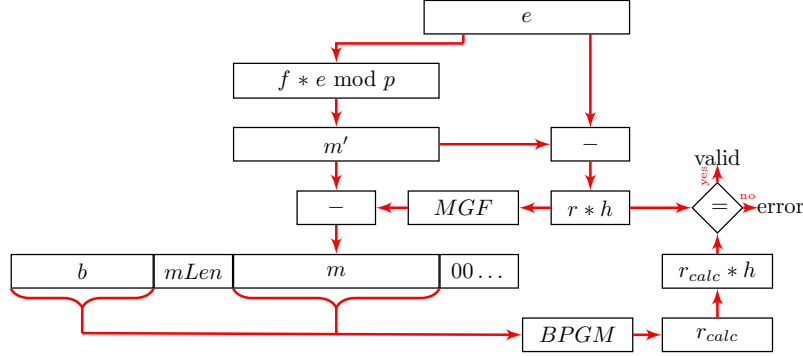


Fig. 2. NTRU decryption with SVES

The NTRU decryption is shown in Fig. 2. It retrieves the original message  $m$  from the ciphertext  $e$  through four steps. The first step extracts  $m'$  by multiplying the ciphertext  $e$  with the private key  $f$  as  $m' = f * e \pmod{p}$ . In the second step,  $r * h$  can be retrieved by subtracting  $m'$  from the ciphertext  $e$ , as the equation  $r * h = e - m'$  holds. The third step uses the resulting product as an input to the MGF to retrieve the padded message as  $m = m' - MGF(r * h)$ . The fourth step checks the validity of the ciphertext by applying the BPGM to the corresponding elements of  $m$  to produce the value  $r_{calc}$ . The multiplication result of  $r_{calc} * h$  is now compared with the polynomial  $r * h$  from the second decryption step. If both polynomials are equal, the algorithm outputs the padded message  $m$ . Otherwise an invalid ciphertext input is detected and the algorithm outputs an error message.

## 4 NTRU Full Hardware Architecture

Our proposed NTRU hardware architecture is illustrated in Fig. 3. The encryption and decryption flows are highlighted in green and red, respectively. To keep the area costs low, the encryption and decryption operation share common hardware modules. The resource sharing is managed by a small controller that sets the data selector values of all multiplexers. The NTRU architecture is composed of four main hardware modules: *CONV*, *BPGM*, *MGF* and *MOD p*. Their implementation is described in the following subsections.



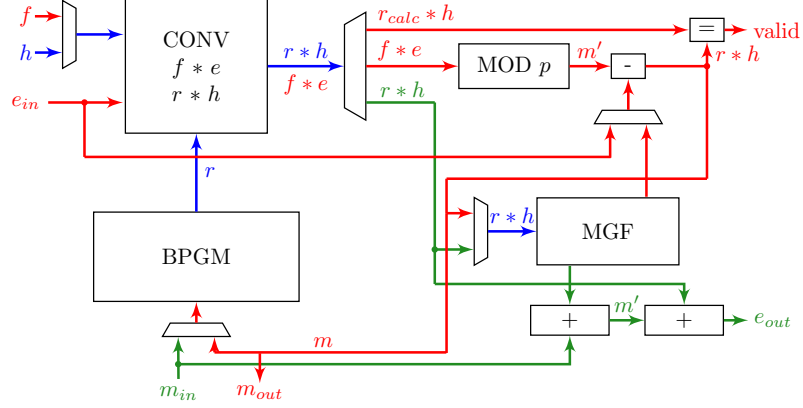


Fig. 3. NTRU architecture, green encryption, red decryption, blue shared

#### 4.1 Convolution (*CONV*)

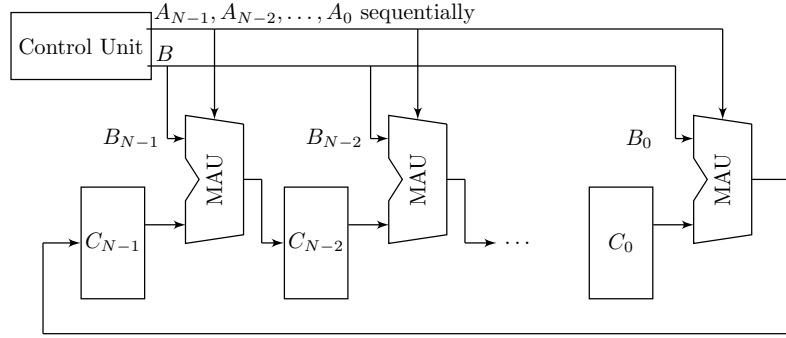


Fig. 4. Circular convolution model

In this work, we adopted the convolution architecture of [15] because of its efficiency and simplicity. This architecture is able to multiply a ternary polynomial with a regular polynomial in  $R_{N,q}$ . However, in order to support the encryption and decryption operations, the following modifications are required: i) integration of the *control unit* to manage the convolution during encryption and decryption operations; and ii) support for the multiplication of two regular polynomials ( $f * e$ ). The enhanced convolution circuit (*CONV*) is shown in Fig. 4.

*CONV* multiplies a ternary polynomial  $A \in \mathcal{T}_N$  with coefficients  $\{-1, 0, 1\}$  and a regular polynomial  $B \in R_{N,q}$ . The circularity of the convolution is realized by shifting the resulting polynomial  $C$  in an LFSR. Depending on the

sequentially inputted coefficient of  $A$ , each Modular Arithmetic Unit ( $MAU$ ) respectively either adds  $B_i$  to  $C_i$ , subtracts  $B_i$  from  $C_i$  or keeps  $C_i$  unchanged, where  $i = 0, 1, \dots, N - 1$ . The result of each  $MAU$  is forwarded to the next register. A more in-depth explanation of the functionality is given in the following: During the first clock cycle, the control logic outputs coefficient  $A_0$ , which is either -1, 0 or 1. The value of  $A_0$  determines the operation mode of all  $MAU$  instances. In case of a -1 a subtraction is performed, in case of a 0 the value of the previous register is forwarded, and in case of a 1 an addition is performed. The first input of the  $i$ -th  $MAU$  is the coefficient  $B_i$ , which remains fixed for all clock cycles. The second input is the value of the previous register  $C_i$ . As  $q = 2048$  each value of  $B_i$  and  $C_i$  has 11 bits. In the next cycle, the coefficient  $A_1$  selects the operation mode of the  $MAUs$ . After  $n$  clock cycles the result of  $A * B$  is stored in the registers.

During encryption, the ternary polynomial  $r$  and a regular polynomial  $h$  are multiplied through  $CONV$ , thus generating  $r * h$ . However, the decryption requires a multiplication of two regular polynomials  $f$  and  $e$ . In order to use  $CONV$  for this multiplication, we use the definition of  $f$  given in the standard, such that  $f = 1 + pF$ , where  $F$  is a ternary polynomial. As a result,  $f * e = (1 + pF) * e = e + pF * e$ . To obtain  $pF * e$ , we repeat the convolution of  $F * e$  two more times ( $p = 3$ ) without resetting the registers after each round. Thus, after  $n$  clock cycles the registers have the value  $F * e$ , after  $2n$  cycles  $F * e + F * e$ , and after  $3n$  cycles  $F * e + F * e + F * e = 3(F * e)$ . At the end of this operation, the *control unit* inputs a ternary polynomial such that the value of the first coefficient is 1 and 0 otherwise. The second input will remain with the polynomial  $e$ . This procedure for calculating the addition of  $pF * e$  with  $e$  takes one round ( $n$  cycles). It is also possible to skip this round if the registers are preloaded with  $e$  at the beginning of the decryption process. In addition to the calculation of  $f * e$ , the decryption has to calculate  $r_{calc} * h$ , which requires one additional round. The proposed process increases the convolution processing time during the decryption operation by a factor of four. However, it avoids the integration of additional multipliers, thus decreasing the required area for the decryption.

#### 4.2 Blinding Polynomial Generation Method ( $BPGM$ )

Hash functions are the core of the  $BPGM$  and  $MGF$  modules. The IEEE-1363.1 standard suggests the use of SHA-1 or SHA-256, depending on the chosen parameter set. SHA-1 and SHA-256 have a 512 bit input and 160 or 256 bit output, respectively. The seed varies in each hash call by using the four values described in Subsection 3.2 ( $OID, m, b, h_{trunc}$ ) concatenated with a *counter* value, which is increased after each hash call. The counter ensures that the hash output changes in each call. Our SHA cores use parts of the open core modules from [20].

The concrete generation of the ternary polynomial  $r$ , which is the output of the  $BPGM$ , is described in the following paragraph. Figure 5 shows the generation of the hash output. The *Control Unit* is responsible for managing the counter, setting the seed and writing the hash output to a buffer. The size of the

buffer depends on the *minCallsR* variable defined in the standard. The buffer output is used for determining the indexes of ones and minus ones in  $r$ . In total, the polynomial  $r$  has  $d_r$  ones and  $d_r$  minus ones. The BPGM uses  $c$ -bits (a variable in the standard) from the buffer output and calculates this value modulo  $N$ . This modulo operation is currently done in a naive way by subtracting  $N$  from the value until it is smaller than  $N$ . Unfortunately, the algorithm used in 4.4 cannot be used because  $N$  is not a Mersenne prime number. However, Barrett and Montgomery reduction can be considered in future works. The result of the  $c$ -bit hash value modulo  $N$  determines the position of the corresponding one or minus one in the polynomial. If the position was already set, the next  $c$ -bits are used to create a new random position. The described process is repeated with new bits from the buffer until all indexes are found.

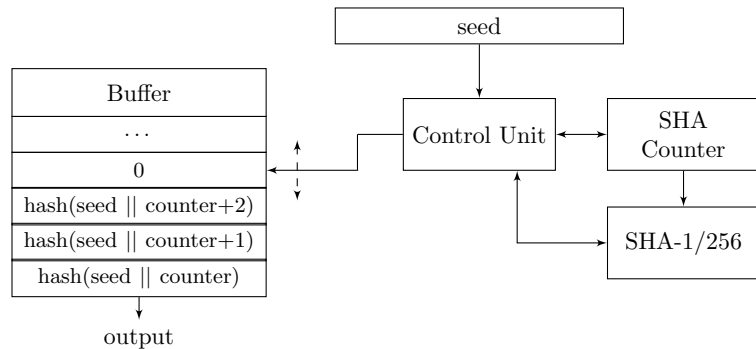


Fig. 5. Buffer generation

### 4.3 Mask Generation Function (*MGF*)

The *MGF* shares the buffer generation module presented in Fig. 5 with the *BPGM*. However, instead of using the buffer output for finding the value of the indexes of  $r$ , the *MGF* transforms the output from a binary into a ternary representation. More specifically, repetitively eight bits are taken from the buffer and converted to five ternary values if the value of these eight bits is smaller than  $3^5 = 243$ . Otherwise, it is rejected and the next byte of the buffer is used. In contrast to the algorithm defined in the standard, we use an efficient Lookup Table (LUT) for this conversion.

### 4.4 Modulo Reduction (*MOD p*)

The naive way to compute the modulo reduction of polynomials is to repeatedly subtract the modulus  $p$  from each coefficient of the polynomial until the coefficient is smaller than  $p$ . However, as  $p = 3$  is a Mersenne prime number,

a faster method to calculate the modulo reduction can be employed, as shown in [12] and optimized in [6]. Algorithm 1 presents the modulo reduction used in our NTRU architecture. To improve the throughput, the  $MOD\ p$  block is instantiated twice.

---

**Algorithm 1:** Mersenne prime modulo division ( $p = 3$ )

---

```

Input: Integer  $a$ 
Result: Integer  $a \bmod 3$ 
additional_reduction = {0, 1, 2, 0, 1, 2}
// reduce  $a$ 
 $a = (a \gg 8) + (a \& 0xFF)$ 
 $a = (a \gg 4) + (a \& 0xF)$ 
 $a = (a \gg 2) + (a \& 0x3)$ 
 $a = (a \gg 2) + (a \& 0x3)$ 
// at this point  $a < 6$ 
a = additional_reduction[a]

```

---

## 5 NTRU HW/SW Co-Design

While “pure” hardware implementations offer high performance solutions they also have several disadvantages such as the loss in flexibility, portability, and the increasing area overhead. In contrast, a “pure” software solution is easier to develop and maintain. It offers a high flexibility and portability, but provides a lower performance. Hybrid solutions that aim to profit from the individual benefits of hardware and software are the basis of the powerful co-design techniques.

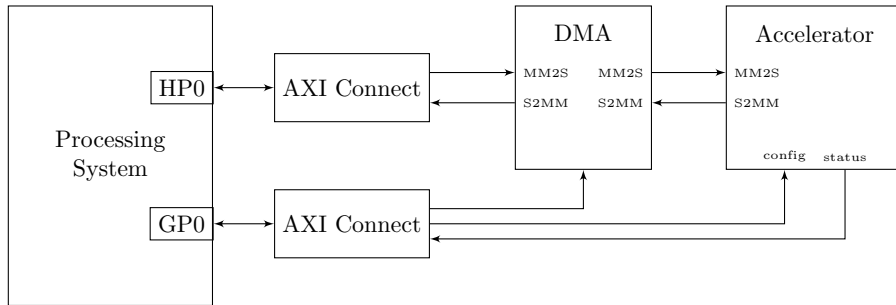
Previous works [6, 21] identify the multiplication as the performance bottleneck of a NTRU software implementation. Therefore, in this work we propose and analyze a HW/SW co-design approach. In our proposal, the NTRU software implementation is accelerated by outsourcing performance-critical multiplications to hardware. This approach can speed up the software implementation while decreasing the hardware costs. As the NTRU key generation must be executed only once in a Public-Key Encryption (PKE) setting, in this work we focus on accelerating the encryption and decryption routines.

System-on-Chip (SoC) FPGAs have become very attractive due to their flexibility and fast development capabilities. They combine one or multiple processors with the programmable logic and are therefore well suited for prototyping the NTRU HW/SW co-design approach. Figure 6 shows the block diagram of our developed architecture. It is composed of three main blocks. The first one is the *Processing System*, which executes the software application. It is linked to the remaining components of the system through the High Performance (HP) and General Purpose (GP) ports, used for high-bandwidth and low-bandwidth data

transfers, respectively. The second block is the *Accelerator*, which performs the polynomial multiplication. Finally, the *Direct Memory Access (DMA)* module is used as a high throughput interface between *Processing System* and *Accelerator* for the data transfer of the polynomial coefficients. These three modules are interconnected through the AXI bus.

In our setting, the *Processing System* is responsible for configuring the *DMA* module. It sets the memory region (start/end address), configures the direction and triggers the data transfer. The configuration is realized by writing to specific registers located in the *DMA* module via the GP0 port. The *Accelerator* has two AXI4-Stream interfaces, which are the optimal solutions to transport arbitrary unidirectional data streams. In order to perform efficient data transfers through the 32-bit data AXI bus structure, one packet contains a value of four coefficients, which are stored in a 32 bit memory line according to  $(000 \parallel A_{i+1} \parallel B_{i+1} \parallel 000 \parallel A_i \parallel B_i)$ , where  $A \in \mathcal{T}_N$ ,  $B \in R_{N,q}$  and  $q = 2048$ .

The *Accelerator* has five different states: *idle*, *read*, *conv\_enc*, *conv\_dec* and *write*. The state switches from *idle* to *read* when the *DMA* sends a valid input packet. The state machine remains in the *read* state until the signal that identifies the last packet is set to one. Depending on the *config* signal, the state machine switches to the *conv\_enc* or *conv\_dec* state. After  $n$  cycles (encryption) or  $4n$  cycles (decryption), the *Accelerator* writes the coefficients via the *DMA* back to the memory. If the *Accelerator* was in the *conv\_dec* state, a modulo reduction is performed on the coefficients of  $B$  before sending the result. The *Accelerator* uses the same sub-modules for the convolution and modulo reduction as described in Subsection 4.1 and Subsection 4.4. More specifically, it uses one *CONV* module and two *MOD p* modules: one for the reduction of  $B_i$  and one for the reduction of  $B_{i+1}$ .



**Fig. 6.** HW/SW co-design architecture with NTRU executed on the 'Processing System' and hardware accelerator connected via DMA module. Abbreviation MM2S indicates memory mapped to stream ports and S2MM stream to memory mapped ports.

## 5.1 Software Implementation

Among the existent NTRU software implementations, for the sake of simplicity and reproducibility, we decided to use the NTRU software implementation presented in the open-source library of NTRUOpenSourceProject [24]. This software solution corresponds to the official NTRU implementation compliant with the IEEE-1363.1 standard and is designed by the NTRU authors.

In our NTRU HW/SW co-design solution, it is possible to switch between software and hardware polynomial multiplication during compile time through define directives.

The software-based multiplication algorithm uses the fact that one of the NTRU multiplication factors always consists of a ternary polynomial, which is characterized by its sparsity (most of the values of the polynomial are zero) and the fact that its coefficients only take the values minus one, zero and one. This allows the substitution of the NTRU multiplication by additions and subtractions of polynomials in  $R_{N,q}$ . The algorithm iterates over all non-zero coefficients of the ternary polynomial and applies operations according to the coefficient's value, i.e. addition in case of a one and subtraction in case of a minus one. The respective index of a coefficient defines the location where these additions/subtractions onto a result array are performed. As coefficients equal to zero are skipped, the overall execution time of the multiplication is decreased.

In addition to the previous single coefficient implementation, the NTRUOpenSourceProject library also provides two multiplication alternatives that are able to perform simultaneously either two (32 bit architecture) or four (64 bit architecture) coefficient addition/subtractions. This modification was introduced by the authors with the aim to support different bit width processing architectures. However, by examining the required clock cycles to perform a multiplication, we observed that on our target architecture Cortex-A53 (with optimization “-O3”), the single coefficient implementation outperforms the other alternatives. As the Cortex-A53 is the core of our platform, we have chosen the single coefficient implementation as the reference implementation used for comparison with our NTRU HW/SW solution.

## 6 Security Analysis

In [16], the authors show an optimization of their NTRU hardware implementation presented in [15]. It is based on scanning the coefficients of the ternary polynomial during the multiplication. When two consecutive zeros are detected, the two corresponding multiplications can be reduced to one, thus speeding up the execution. In the following subsections, we describe the optimized architecture and present the security vulnerability caused by this optimization.

### 6.1 Optimized Architecture

The optimized architecture is able to detect two consecutive zeros in the ternary input polynomial  $A$  (Fig. 4). The processing of a zero coefficient during the

convolution can be seen as a single circular shift of the coefficients  $C_i$ . Therefore, two zeros can be substituted by a single shift of two places within one clock cycle. This results in a reduction of one cycle in the total multiplication time for each pair of consecutive zeros. The implementation of Liu *et al.* [16] requires an additional multiplexer for the MAU, which is connected to the preceding register output of the result coefficient  $C_i$ . In comparison to their original and non-optimized implementation in [15], the authors report a reduction of 36.7% of the execution time for the convolution with the parameter set *ees541ep1*.

## 6.2 Vulnerabilities

In the conference version of this paper [4], it was shown that the optimized implementation of [16] leaks information about the secret key through a timing side-channel because the convolution time depends on the structure of the key, i.e. the amount of double zeros in the secret key polynomial. The term double zero is used in the following to express that two subsequent coefficients of the secret polynomial are zero. Three consecutive zeros are processed as one double zero (the first two) and one single zero (the third coefficient) as opposed to a single zero and then a double zero. This corresponds to the actual optimization in [16]. The vulnerability to a timing side-channel is inherent to the design as the optimization is solely based on exploiting the occurrence of double zeros: The overall reduced processing time is a clear indicator of how many double zeros are present in the secret polynomial  $F$ . Based on this knowledge, part of the key space can be discarded by an attacker. This reduces the effective length of the private key polynomial and hence the security level.

Whereas, the previous conference version [4] provided a bound for the number of possible secret polynomials given an observed amount of double zeros, now the result has been sharpened. The following theorem provides the exact amount of these polynomials, such that the precise complexity reduction of the exhaustive search space can be computed.

**Theorem 1.** *The number of valid ternary polynomials for a given number of double zeros  $d_z$  is given by*

$$u_r(n, d_f, d_z) = \frac{(2d_f + d_z)!}{d_z! \cdot d_f! \cdot d_f!} \cdot \binom{2d_f + 1}{d_s}, \quad (4)$$

where  $n$  is the number of coefficients of the polynomial,  $d_f$  the number of 1's and also of the  $-1$ 's in the private key  $F$ , and  $d_s$  is the amount of single zeros, i.e.  $d_s = n - 2d_z - 2d_f$ . Note that  $2d_f$  is for  $d_f$  1's and the additional  $d_f$   $-1$ 's.

*Proof.* The proof of this theorem consists of two steps as visualized in Fig. 7. It is derived how many possibilities exist to build a polynomial given the amount of 1's,  $-1$ 's, and double zeros.

*Step I:* First all the 1's,  $-1$ 's, and double zeros are placed. A double zero is not to be handled as two numbers, but as one symbol, because both zeros are placed at once. Overall there are  $2d_f + d_z$  spots for these symbols (the grey squares in Fig. 7). This results in

$$\frac{(2d_f + d_z)!}{d_z! \cdot d_f! \cdot d_f!} \quad (5)$$

possibilities to arrange all the 1's,  $-1$ 's and double zeros according to the multiset permutation formula. The denominator is needed to exclude configurations which cannot be distinguished. For example, interchanging the 1's does not change the overall sequence.

*Step II:* Given any configuration of Step I, now the single zeros are placed among the sequence of 1's,  $-1$ 's, and double zeros. However, two conditions have to be satisfied in Step II:

- (a) No two single zeros are allowed next to each other, because they would then count as one double zero.
- (b) A single zero followed by a double zero (000) is not valid as this would in fact be considered as a double zero followed by a single zero (000). This corresponds to the definition of double zeros and matches how zeros are processed in the optimized architecture.

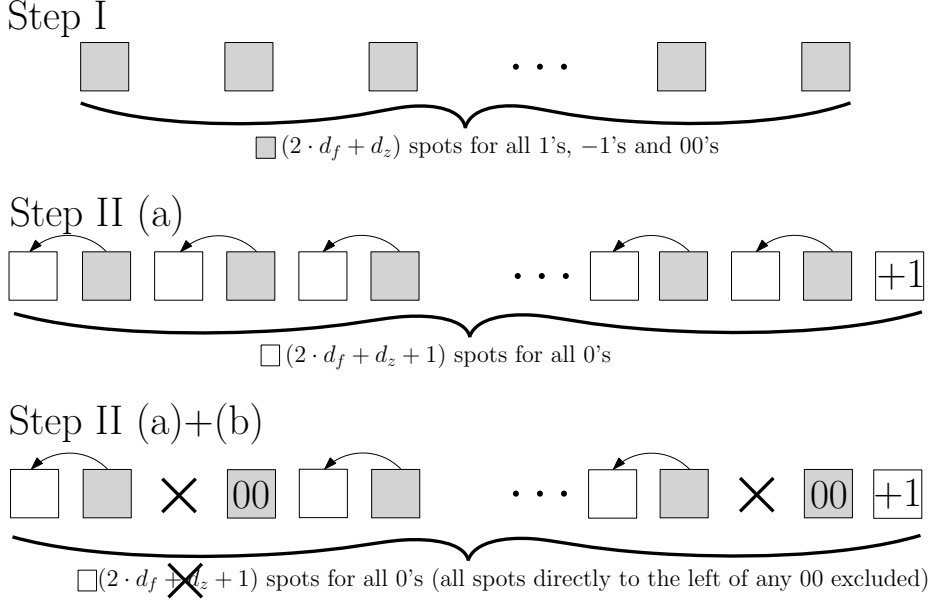
If only condition (a) is considered, there are  $2d_f + d_z + 1$  positions to place the single zeros. This is depicted in Fig. 7 Step II (a) by the white squares: Each of the  $d_f$  1's, the  $d_f$   $-1$ 's, and the  $d_z$  double zeros (i.e. each grey square) has exactly one empty spot to its left ( $d_f + d_f + d_z$ ). Exactly one spot is necessary to ensure that no two single zeros can be placed directly next to each other, that is: no two white squares are next to each other, as there is always a grey square in between. Furthermore, an empty spot is merely a possible position for a single zero. Hence, it is not necessary to actually put a single zero there. Finally, there is an empty spot on the very right (+1), i.e. the white box with +1.

Under condition (b) however, the amount of valid spots for a single zero is limited as the spots left to a double zero are not allowed. In Fig. 7 Step II (a)+(b), they are represented by a crossed-out spot left of every double zero position (grey box with 00). (0 00 is by definition not valid and would be understood as 00 0). Thus, in fact there are only  $2d_f + 1$  free and valid spots (white boxes) for a single zero. There are exactly  $d_z$  double zeros and wherever they are placed, they have got exactly one possible free spot as left neighbour which has to be crossed out. Out of the remaining  $2d_f + 1$  possible positions now  $d_s$  spots are chosen for the  $d_s$  single zeros:

$$\binom{2d_f + 1}{d_s} \quad (6)$$

Because the amount of possibilities in Step (II) holds for any possible configuration in Step (I), multiplying the two intermediate results proves Theorem 1.  $\square$





**Fig. 7.** Visualization of the proof

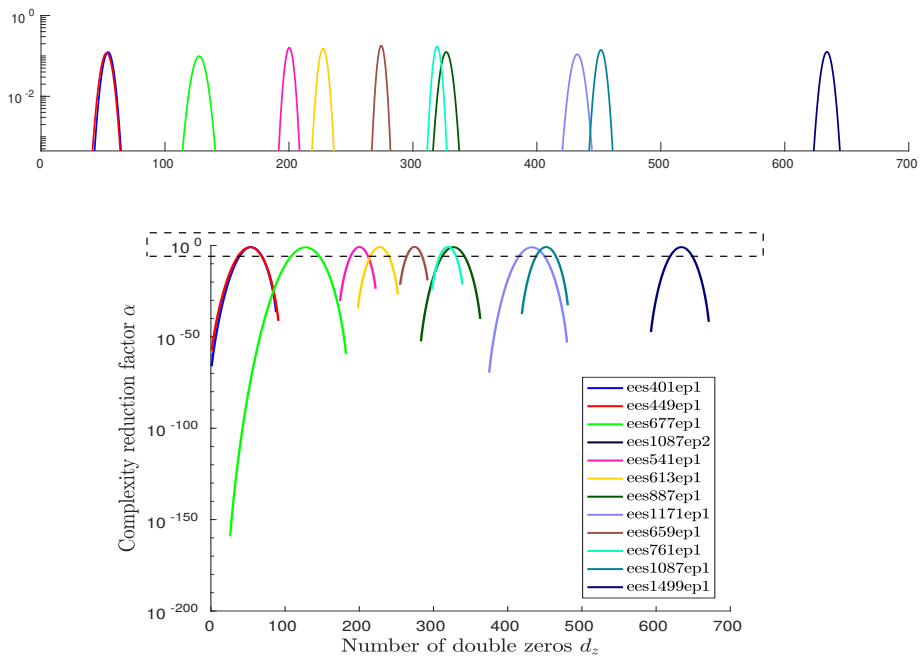
Based on this result, one can compute by which factor the search complexity is reduced given a certain amount of observed double zeros and different parameter sets. Figure 8 illustrates the complexity reduction factor  $\alpha$  for the parameter sets in [11]. This factor can be computed by

$$\alpha = \frac{u_r(n, d_f, d_z)}{K_c} , \text{ with } K_c = \frac{n!}{(d_f!)^2 (n - 2d_f)!} , \quad (7)$$

where  $K_c$  denotes the cardinality of the key space. Overall, for any of the parameter sets as given in [11], the greatest loss of security is given for the minimum amount of double zeros for the respective parameter set. Note that this minimum number of double zeros  $d_{z,min}$  is not necessarily 0, but

$$d_{z,min} = \max \left( \left\lfloor \frac{n - 4d_f}{2} \right\rfloor, 0 \right) . \quad (8)$$

The worst case complexity reduction for every parameter set can be computed given  $d_{z,min}$  by means of Eq. 7. The parameter set with the greatest loss in security when exploiting double zeros as in [16] is *ees677ep1* with a complexity reduction factor up to  $10^{-160}$ . This means that if the secret polynomial has the minimum amount of double zeros, then an attacker can discard most of the key space and only focus on a subset of polynomials which is  $10^{160}$  times smaller than the original key space. Additionally, the enlarged part of Fig. 8 illustrates that a minimum leakage regardless of the amount of double zeros exists. Even for a



**Fig. 8.** Complexity reduction of the exhaustive search space of the private key  $F$  for a known amount of double-zeros and the parameter sets in [11]. The top part of the figure is an excerpt of the whole graph to illustrate complexity reduction in the best case scenario.

configuration with the most likely amount of double zeros, part of the key space can be excluded by the attacker. E.g. for *ees659ep1*, 276 double zeros result in an effective key space with a size that is roughly 17.8% of the whole key space. Consequently, in the best case from a legitimate user’s point of view still 82.2% of the possible secret keys can be neglected by an attacker. In case of every other parameter set, even more keys are discarded in the best case scenario.

Yet, it could be argued that despite the possibility for an attacker to disregard a certain part of the key space, the remaining key space still might be large enough to withstand an exhaustive search. However, this vulnerability only considers brute-force attacks. If by means of a more sophisticated attack the key space shrinks further, security might no longer be guaranteed. This shows that leaking the exact amount of double zeros by a timing side-channel reduces the security. As a final remark, one could consider exploiting exactly the minimum amount of double zeros for a given parameter set. Because given a parameter set there are at least  $d_{z,min}$  double zeros, the attacker gains no information if for any  $d_z$  exactly  $d_{z,min}$  double zeros are used to speed up the computations. Consequently, a counter could keep track of how many double zeros have been exploited by processing them as double zero. If  $d_{z,min}$  double zeros have been used for a speed-up, the zeros to come are processed normally. For most of the parameter sets, the amount of clock cycles can be reduced. However, the usage of a counter introduces a new side-channel. If an attacker can detect the point in time when the implementation stops processing double zeros, the key space once again can be reduced. Furthermore, an attacker might be able to detect which coefficients are double zeros, resulting in an additional complexity reduction. Overall, saving few clock cycles in such a way is not worth the overhead and especially the leakage through this side-channel.

## 7 Results

Our two proposed NTRU architectures (full HW and HW/SW approach) were implemented on the Xilinx Zynq UltraScale+ MPSoC ZCU102 platform, which contains among others a quad-core ARM Cortex-A53. The full hardware solution only uses the programmable logic of the platform whereas the HW/SW design utilizes the ARM Cortex-A53, too.

### 7.1 Results of Full Hardware Implementation

The IEEE-1363.1 standard defines different parameter sets for different security levels and optimization goals. Table 1 summarizes the results of our proposed full hardware implementation. It contains the total number of LUTs, registers, and the required number of clock cycles for encryption and decryption. The results show that the number of LUTs scales with the parameter  $n$ , which determines the size of the polynomials.

Figure 9 provides a more detailed view of the required clock cycles for the encryption. The time required for the convolution depends directly on the value

**Table 1.** Clock cycle count and resource utilization of the full hardware implementation with parameter sets defined in IEEE-1363.1

Security Level	Parameter Set	$n$	LUT	Register	#CC Enc.	#CC Dec.
Low	<i>ees401ep1</i>	401	29,119	25,445	3,423	5,430
	<i>ees541ep1</i>	541	36,990	27,617	2,409	5,116
	<i>ees659ep1</i>	659	45,685	31,540	2,413	5,711
Middle	<i>ees449ep1</i>	449	32,851	27,263	3,642	5,890
	<i>ees613ep1</i>	613	44,168	30,555	2,675	5,743
	<i>ees761ep1</i>	761	52,208	35,586	2,799	6,606
High	<i>ees677ep1</i>	677	50,441	38,819	4,020	7,407
	<i>ees887ep1</i>	887	64,539	42,636	3,113	7,551
	<i>ees1087ep1</i>	1,087	74,774	50,000	3,760	9,197
Highest	<i>ees1087ep2</i>	1,087	75,393	53,740	4,723	10,159
	<i>ees1171ep1</i>	1,171	74,730	52,497	4,345	10,202
	<i>ees1499ep1</i>	1,499	98,774	66,200	4,715	12,212

of  $n$  because  $n$  clock cycles are required for the circular shift within the LFSR. Results show that the impact of the padding scheme on the cost and performance of NTRU is not negligible. For some NTRU configurations, in hardware the convolution has a minor influence on the computation cost when compared to the SVES padding scheme. For the parameter set *ees401ep1*, the padding scheme takes nearly 90% of the encryption time.

The main bottleneck of the padding scheme are the repetitive calls of the hash function. The number of clock cycles spent by the BPGM mainly depends on the parameter  $d_r$ , which determines the number of ones and minus ones in  $r$  and thus the number of hash calls. Please note that the amount of clock cycles for the BPGM slightly depends on the seed of the hash function. As described in Subsection 4.2, the  $c$ -bits of the hash value must be discarded if the generated index is already set. Moreover, the runtime of the reduction modulo  $N$  varies. For these two reasons, a variation in the execution time of the BPGM in the lower double-digit range was observed. As the differences are small, the determined values remain meaningful.

Figure 10 illustrates the computation costs for the decryption. Whereas the decryption requires  $4n$  clock cycles for the convolution, the encryption only requires  $n$  clock cycles. Both, the convolution and the modulo  $p$  operation directly scale with  $n$ . Note that the execution time of the *MGF* can also slightly vary when the fetched byte is not smaller than 243. In practice, this has only a marginal influence on the performance.

## 7.2 Results of HW/SW Co-Design

In this subsection, the resource utilization and runtime of the HW/SW co-design is described. For all clock cycle measurements, the Cortex ARM-A53 ran at a

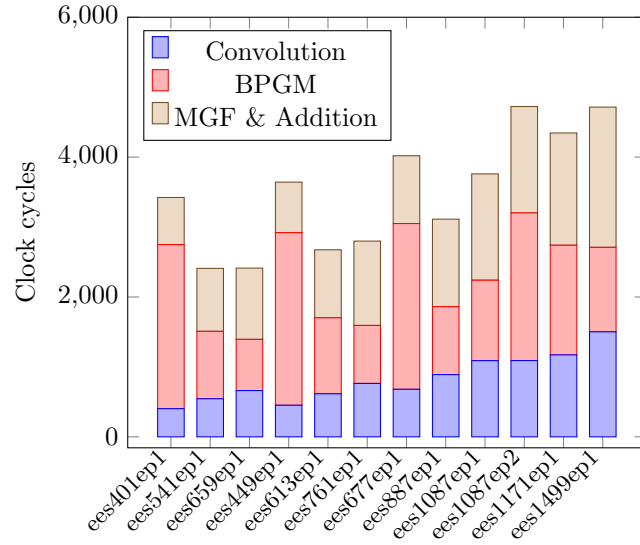


Fig. 9. Clock cycles for encryption of the full hardware implementation

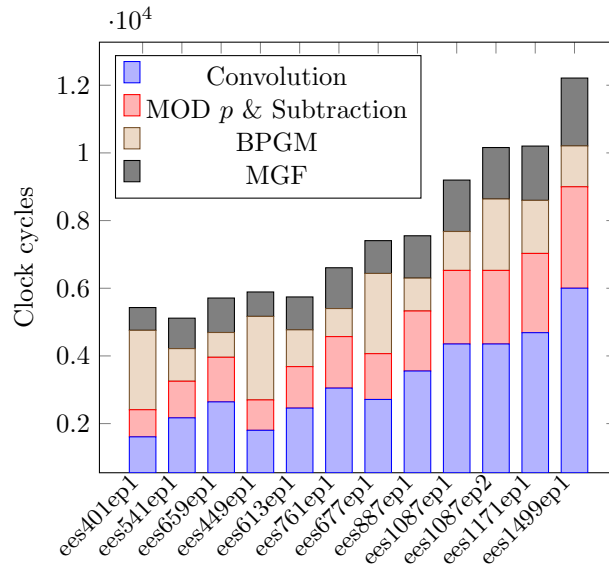


Fig. 10. Clock cycles for decryption of the full hardware implementation

target frequency of 1,200 MHz (real 1,199.880 MHz) and the hardware accelerator at a target frequency of 200 MHz (real 187.481 MHz). The runtime was measured using the cycle count register provided by the Performance Monitor Unit (PMU) of the Cortex ARM-A53. The correctness of these measurements was cross-checked with a hardware counter.

**Area.** Table 2 presents the resource utilization of our HW/SW solution. The results show that the amount of LUTs and registers depends on the size of the processed polynomials. The largest components in the design are the *Accelerator* and the *DMA* module. The *DMA* module requires, in addition to the listed LUTs and registers, two Block RAM (BRAM) instances. Its resource utilization remains constant for all parameter sets. Within the *Accelerator*, the *CONV* module is the largest component. The results also show that the required amount of LUTs and registers is lower for the HW/SW solution when compared to the full hardware design.

**Table 2.** HW/SW co-design resource utilization

Parameter Set	Total LUT/Register	Accelerator LUT/Register	Convolution LUT/Register	DMA LUT/Register
<i>ees401ep1</i>	19,582/16,104	15,572/10,710	11,378/5,342	1,263/1,759
<i>ees541ep1</i>	25,108/19,729	21,090/14,335	15,167/7,074	1,273/1,759
<i>ees659ep1</i>	29,181/22,787	25,169/17,393	18,454/8,614	1,270/1,759
<i>ees449ep1</i>	21,463/17,375	17,440/11,981	12,763/5,980	1,272/1,759
<i>ees613ep1</i>	27,547/21,583	23,533/16,189	17,169/8,013	1,265/1,759
<i>ees761ep1</i>	32,615/25,449	28,602/20,055	21,303/9,944	1,267/1,759
<i>ees677ep1</i>	29,891/23,249	25,880/17,855	18,956/8,848	1,267/1,759
<i>ees887ep1</i>	38,670/28,752	34,659/23,358	24,832/11,590	1,268/1,759
<i>ees1087ep1</i>	46,788/33,966	42,773/28,572	30,423/14,199	1,271/1,759
<i>ees1087ep2</i>	46,788/33,966	42,773/28,572	30,423/14,199	1,271/1,759
<i>ees1171ep1</i>	50,402/36,183	46,384/30,789	32,757/15,296	1,272/1,759
<i>ees1499ep1</i>	63,221/44,766	59,197/39,372	41,912/19,574	1,276/1,759

**Performance.** Table 3 and Table 4 present the clock cycle counts for the NTRU software implementation with and without hardware accelerator. For the software implementation, the open-source library discussed in Section 5.1 is used. Two different compiler optimization levels were tested: '-O1' for small optimizations (Table 3) and '-O3' for the highest speed optimization (Table 4). The results with the hardware accelerator include the communication overhead. The measured clock cycles are related to the clock of the *Processing System*, i.e. one clock cycle in hardware corresponds to roughly six cycles in the *Processing System*.

The defined parameter sets have different optimization goals. The parameter sets *ees401ep1*, *ees449ep1*, *ees677ep1* and *ees1087ep2* are optimized for size and have the smallest polynomials in their respective security category. The sets *ees541ep1*, *ees613ep1*, *ees887ep1* and *ees1171ep1* are cost-optimized and have the lowest value of '(operation time)<sup>2</sup> × size'. Obviously, *ees659ep1*, *ees761ep1*, *ees1087ep1* and *ees1499ep1* are optimized for speed. They have the lowest amount of clock cycles in their security category. This classification does not apply for the HW/SW co-design anymore. Similar to the area consumption, the required amount of clock cycles for the polynomial multiplication depends on the parameter  $n$ .

With the optimization level '-O1' and the fastest parameter set of the highest security category—parameter set *ees1499ep1*—a speedup factor of 44.38 can be achieved for calculating  $h * r$  and a factor of 17.74 for calculating  $peF + e \bmod p$ . By setting the optimization flag to '-O3', the speedup decreases to 9.87 and 4.77, respectively. However, this is still a considerable improvement because the Cortex-A53 is already a very powerful processor running at a higher clock frequency when compared to the frequency of the hardware accelerator. The runtime of the whole encryption function is improved by the factor 5.55 ('-O1') and 1.99 ('-O3'), the runtime for decryption is improved by a factor of 7.94 ('-O1') and 2.60 ('-O3'). Other parameter sets even have a higher improvement because the mentioned parameter set belongs to the fastest in software.

**Table 3.** HW/SW co-design cycle count (kilo cycles) when optimization -O1 is used ('yes' and 'no' for usage of hardware accelerator)

Parameter Set	$h * r$ no/yes	$peF + e$ no/yes	Encryption no/yes	Decryption no/yes
<i>ees401ep1</i>	557.7/13.6	572.1/27.2	690.6/145.8	1,278.2/188.5
<i>ees541ep1</i>	329.2/14.4	348.9/32.9	440.8/125.6	809.3/178.1
<i>ees659ep1</i>	312.3/16.0	336.4/39.3	433.2/136.0	772.6/178.7
<i>ees449ep1</i>	738.0/15.5	754.3/30.8	889.3/166.6	1,662.1/214.0
<i>ees613ep1</i>	417.0/15.5	439.2/36.8	541.9/140.6	1,004.1/199.4
<i>ees761ep1</i>	397.0/18.0	424.4/44.3	537.5/157.7	966.4/207.2
<i>ees677ep1</i>	1,296.2/20.1	1,320.9/43.0	1,494.8/218.8	2,844.0/289.3
<i>ees887ep1</i>	879.0/21.5	911.3/52.2	1,065.5/207.6	2,007.6/289.0
<i>ees1087ep1</i>	841.0/24.6	880.4/62.4	1,049.8/232.5	1,937.7/303.2
<i>ees1087ep2</i>	1,588.6/26.3	1,628.1/64.0	1,830.0/268.0	3,499.9/372.7
<i>ees1171ep1</i>	1,513.0/27.3	1,555.0/67.9	1,761.0/276.0	3,360.9/388.5
<i>ees1499ep1</i>	1,446.8/32.6	1,500.8/84.6	1,725.3/310.6	3,238.3/407.8

**Table 4.** HW/SW co-design cycle count (kilo cycles) when optimization -O3 is used ('yes' and 'no' for usage of hardware accelerator)

Parameter Set	$h * r$ no/yes	$peF + e$ no/yes	Encryption no/yes	Decryption no/yes
<i>ees401ep1</i>	141.6/13.0	153.3/24.3	257.3/131.5	421.5/164.0
<i>ees541ep1</i>	81.1/14.0	97.0/29.2	180.1/114.8	289.1/154.3
<i>ees659ep1</i>	73.4/16.0	92.8/34.7	181.2/124.8	273.6/156.0
<i>ees449ep1</i>	183.7/14.6	196.3/27.2	316.5/149.9	524.5/186.5
<i>ees613ep1</i>	100.5/15.3	118.1/32.8	211.8/128.4	343.2/171.8
<i>ees761ep1</i>	90.6/17.6	113.7/39.3	215.3/144.8	329.6/182.7
<i>ees677ep1</i>	307.1/19.6	327.7/38.5	482.0/194.7	824.5/247.9
<i>ees887ep1</i>	201.7/21.2	229.1/47.0	365.9/187.5	613.1/249.8
<i>ees1087ep1</i>	189.5/24.5	221.4/55.2	374.8/212.1	593.9/263.9
<i>ees1087ep2</i>	357.1/25.6	389.4/56.8	567.8/239.5	980.8/318.5
<i>ees1171ep1</i>	335.1/26.5	369.3/60.2	553.0/247.8	947.1/330.6
<i>ees1499ep1</i>	314.8/31.9	358.8/75.2	559.2/280.7	919.7/353.8

## 8 Conclusion

Efficient and secure post-quantum cryptography is mandatory to ensure long-term security. The lattice-based cryptographic scheme NTRU is a promising candidate to replace traditional PKC. Previous works in NTRU hardware implementations focused on the development of a fast polynomial multiplication architecture. In this work, for the first time, we propose a full hardware/software implementation that is compliant with the IEEE-1363.1 standard. By including the SVES scheme, our NTRU solution is secure against CCA. The results show that for the full NTRU hardware implementation, the costs of the SVES scheme cannot be neglected. For some parameters it requires nearly 90 % of the total encryption time. In order to increase the flexibility, the HW/SW co-design NTRU solution can be used. By outsourcing the polynomial multiplication and modulo operation to hardware, we achieve significant performance improvements. Moreover, we show that an efficient NTRU implementation is not enough. We have demonstrated that a state-of-the-art hardware optimized NTRU [16] presents a leakage that reduces the security level of NTRU. Their polynomial multiplication introduces a timing side-channel that reduces the private key search space. Thus an attacker can exploit this weakness to recover the secret key.

## References

1. Atici, A.C., Batina, L., Fan, J., Verbauwhede, I., Örs, S.B.: Low-cost implementations of NTRU for pervasive security. In: 19th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2008, July 2-4, 2008, Leuven, Belgium. pp. 79–84 (2008)



2. Bailey, D.V., Coffin, D., Elbirt, A., Silverman, J.H., Woodbury, A.D.: NTRU in Constrained Devices. In: CHES. pp. 262–272 (2001)
3. Bailey, D.V., Coffin, D., Elbirt, A.J., Silverman, J.H., Woodbury, A.D.: NTRU in constrained devices. In: Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings. pp. 262–272. No. Generators (2001)
4. Braun, K., Fritzmann, T., Maringer, G., Schamberger, T., Sepúlveda, J.: Secure and compact full ntru hardware implementation. In: 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). pp. 89–94. IEEE (2018)
5. Collen Marie, O.: Efficient NTRU implementation. Master’s thesis, Worcester Polytechnic Institute (2002), <https://www.wpi.edu/Pubs/ETD/Available/etd-0430102-111906/unrestricted/corourke.pdf>
6. Guillen, O.M., Pöppelmann, T., Mera, J.M.B., Bongenaar, E.F., Sigl, G., Sepulveda, J.: Towards post-quantum security for IoT endpoints with NTRU. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2017. pp. 698–703 (March 2017). <https://doi.org/10.23919/DATE.2017.7927079>
7. Hoffstein, J., Pipher, J., Schanck, J.M., Silverman, J.H., Whyte, W., Zhang, Z.: Choosing parameters for NTRUEncrypt. IACR ePrint **2015**, 708 (2015), <http://eprint.iacr.org/2015/708>
8. Howgrave-Graham, N., Nguyen, P.Q., Pointcheval, D., Proos, J., Silverman, J.H., Singer, A., Whyte, W.: The impact of decryption failures on the security of NTRU encryption. In: Annual International Cryptology Conference. pp. 226–246. Springer (2003)
9. Howgrave-Graham, N., Silverman, J.H., Singer, A., Whyte, W.: NAEP: Provable security in the presence of decryption failures. IACR Cryptology ePrint Archive **2003**, 172 (2003)
10. Hu, F., Wilhelm, K., Schab, M., Lukowiak, M., Radziszowski, S., Xiao, Y.: Ntru-based sensor network security: a low-power hardware implementation perspective. Security and Communication Networks **2**(1), 71–81 (2009)
11. IEEE: IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices. IEEE Std 1363.1-2008 pp. C1–69 (March 2009). <https://doi.org/10.1109/IEEESTD.2009.4800404>
12. Jones, D.W.: Modulus without division, a tutorial (2001), <http://homepage.cs.uiowa.edu/~jones/bcd/mod.shtml>
13. Kamal, A.A., Youssef, A.M.: An FPGA implementation of the NTRUEncrypt cryptosystem. In: Microelectronics (ICM), 2009 International Conference on. pp. 209–212. IEEE (2009)
14. Lee, M.K., Kim, J.W., Song, J.E., Park, K.: Sliding window method for NTRU. In: ACNS. pp. 432–442 (2007)
15. Liu, B., Wu, H.: Efficient architecture and implementation for NTRUEncrypt system. In: Circuits and Systems (MWSCAS), 2015 IEEE 58th International Midwest Symposium on. pp. 1–4. IEEE (2015)
16. Liu, B., Wu, H.: Efficient multiplication architecture over truncated polynomial ring for NTRUEncrypt system. In: IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016. pp. 1174–1177 (2016)
17. L.K., G.: A fast quantum mechanical algorithm for database search. In: 28th Annual ACM Symposium on the Theory of Computing. p. 212 (May 1996)
18. Monteverde, M.: NTRU Software Implementation for Constrained Devices. Master’s thesis, Katholieke Universiteit Leuven (2008)

19. National Institute of Standards and Technology: Announcing request for nominations for public-key post-quantum cryptographic algorithms (2016), <https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms>
20. de la Piedra, A.: SHA-256 Core (2013), <https://opencores.org/project/sha256core/>
21. Sepulveda, J., Zankl, A., Mischke, O.: Cache attacks and countermeasures for ntruencrypt on mpsoCs: Post-quantum resistance for the iot. In: 2017 30th IEEE International System-on-Chip Conference (SOCC). pp. 120–125 (Sep 2017). <https://doi.org/10.1109/SOCC.2017.8226020>
22. Sepulveda, J., Liu, S., Mera, J.M.B.: Post-quantum enabled cyber physical systems. In: IEEE Embedded Systems Letters. pp. 1–4 (2019)
23. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. pp. 124–134. Ieee (1994)
24. Whyte, W.: NTRU Open Source Project (2017), <https://github.com/NTRUOpenSourceProject/NTRUEncrypt>
25. Zhan, X., Zhang, R., Xiong, Z., Zheng, Z., Liu, Z.: Efficient Implementations of NTRU in Wireless Network. *Communications and Network* **5**(03), 485 (2013)