Towards Reliable and Secure Post-Quantum Co-Processors based on RISC-V

Tim Fritzmann*, Uzair Sharif*, Daniel Müller-Gritschneder*, Cezar Reinbrecht†,
Ulf Schlichtmann*, Johanna Sepulveda*
*Technical University of Munich, Munich, Germany
†Delft University of Technology, Delft, Netherlands
{tim.fritzmann, uzair.sharif, daniel.mueller, ulf.schlichtmann, johanna.sepulveda}@tum.de
c.r.wedigreinbrecht@tudelft.nl

***Abstract*—Increasingly complex and powerful Systems-on-Chips (SoCs), connected through a 5G network, form the basis of the Internet-of-Things (IoT). These technologies will drive the digitalization in all domains, e.g. industry automation, automotive, avionics, and healthcare. A major requirement for all above domains is the long-term (10 to 30 years) secure communication between the SoCs and the cloud over public 5G networks. The foreseeable breakthrough of quantum computers represents a risk for all communication. In order to prepare for such an event, SoCs must integrate secure quantum-computer-resistant cryptography which is reliable and protected against SW and HW attacks. Empowering SoCs with such strong security poses a challenging problem due to limited resources, tight performance requirements and long-term life-cycles. While current works are focused on efficient implementations of post-quantum cryptography, implementation-security and reliability aspects for SoCs are still largely unexplored. To this end, we present three contributions. First, we present a RISC-V co-processor for post-quantum security, able to support lattice-based cryptography. Second, we use HW/SW co-design techniques to accelerate the NTT transformation and hash generation. Third, we perform the fault analysis of the implementation. We show that our co-processor achieves high reliability and security capabilities while preserving good performance.**

***Index Terms*—Lattice-based cryptography, NewHope, RISC-V, HW/SW co-design**

I. Introduction

Traditional Public-Key Cryptography (PKC) provides the basis for establishing secured communication channels between multiple parties. By using hard-to-solve mathematical problems, such as factoring large integers (RSA) or computing discrete logarithms (ECC), PKC ensures confidentiality, authenticity and non-repudiation of electronic communications and data storage. However, the foreseeable breakthrough of quantum computers represents a risk for many PKC ecosystems. It has been shown that cryptography based on these mathematical problems will be broken in polynomial time by Shor's algorithm, once a sufficiently large quantum computer is built. Hence, the integration of quantum-resistant (also called post-quantum) cryptography becomes mandatory to achieve long term security. For this, there exist mathematical problems which remain secure even in the presence of quantum computers. Among the different post-quantum alternatives, lattice-based cryptography, based on computationally hard problems in certain lattices, is an efficient solution. However, empowering electronic devices with novel and strong security, poses a challenging problem due to the on-time market pressure, limited resources and tight performance requirements. Specially, when dealing with complex Systems-on-Chips (SoCs) that integrate several heterogeneous IP hardware components from different providers. Open source hardware opens a path towards an ultra-fast design cycle for complex and highly customized SoCs. RISC-V is a free and open instruction set architecture (ISA), which enables a new era of processor innovation. RISC-V allows the development of open source hardware with hardware security extensions and secure co-processors.

In this work, we propose a HW/SW co-design for lattice-based cryptography based on RISC-V. In summary, our contributions are: i) first implementation of lattice-based cryptography on an energy-efficient RISC-V core; ii) first real HW/SW co-design for lattice-based cryptography for high flexibility but also efficiency; iii) acceleration of the two performance bottlenecks: the Number Theoretic Transform (NTT) and hash generation; and iv) fault analysis of the algorithm based on simulation-based fault injection.

II. Lattice-Based Cryptography

Most lattice-based cryptographic schemes are built upon the Learning With Errors (LWE) problem and its variants. Due to its efficiency and practicality, the Ring-Learning With Errors (R-LWE) problem, introduced in [1], is the most popular LWE variant. It has shown to be as hard as the NP-hard approximate Shortest Vector Problem (SVP) in a regular lattice.

Cryptographic schemes based on the R-LWE problem require polynomial additions and multiplications, where all calculations are performed in the ring $\mathcal{R} = \mathbb{Z}_q/\langle x^n+1\rangle$. While polynomial additions can be performed in $\mathcal{O}(n)$ operations, a direct approach for the polynomial multiplication requires $\mathcal{O}(n^2)$ operations. An R-LWE instance can be created by calculating $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$, where $\boldsymbol{a}$ is a publicly known polynomial, $\boldsymbol{s}$ the secret polynomial and $\boldsymbol{e}$ an error polynomial. The hardness of the R-LWE problem is to recover the secret polynomial from $\boldsymbol{c}$ or to distinguish $\boldsymbol{c}$ from uniform noise. If the degree of the polynomials and/or the coefficients of the error polynomial are large, it is unfeasible for an attacker to solve this problem.

978-3-9819263-2-3/DATE19/©2019 EDAA

Alice (server):		**Bob (client):**
① seed $\xleftarrow{\$} \{0,1\}^{256}$		
$\boldsymbol{a} \leftarrow$ Gen_a(seed)		
② $\boldsymbol{s}, \boldsymbol{e} \xleftarrow{\$} \chi$		② $\boldsymbol{s'}, \boldsymbol{e'}, \boldsymbol{e''} \xleftarrow{\$} \chi$
③ $\boldsymbol{b} \leftarrow \boldsymbol{as} + \boldsymbol{e}$	$\xrightarrow{\boldsymbol{b}, \text{seed}}$	③ $\boldsymbol{a} \leftarrow$Gen_a(seed)
		④ $m \xleftarrow{\$} \{0,1\}^{256}$
		$\boldsymbol{d} \leftarrow$ NHEncode(m)
		⑤ $\boldsymbol{u} \leftarrow \boldsymbol{as'} + \boldsymbol{e'}$
⑤ $\boldsymbol{d'} \leftarrow \boldsymbol{c} - \boldsymbol{us}$	$\xleftarrow{\boldsymbol{u}, \boldsymbol{c}}$	$\boldsymbol{c} \leftarrow \boldsymbol{bs'} + \boldsymbol{e''} + \boldsymbol{d}$
$m' \leftarrow$ NHDecode($\boldsymbol{d'}$)		

Protocol 1. NewHope R-LWE key/message encapsulation. All polynomials in $\mathcal{R}$ are printed in bold and non-bold letters are used for usual 256 bit-arrays. Let $\xleftarrow{\$}$ denote the sampling process.

A. NewHope

Among all the lattice-based cryptographic schemes, NewHope is due to its simplicity and performance one of the most promising candidates. The original scheme was proposed in [2] (NewHope-USENIX). In 2017, NewHope was revised and submitted to the NIST competition [3] (NewHope-NIST). Protocol 1 shows the underlying algorithm of NewHope-NIST. Five steps are highlighted due to the relevance to the present work. For more details, we refer the reader to [3].

In Step 1, Alice takes 32 bytes from a physical source of entropy and stores them as a seed. The random bytes are seen as an input of the protocol. The seed is used to generate the public polynomial $\boldsymbol{a}$ with the function Gen_a, which expands the seed using the SHAKE-128 hash function. The coefficients of $\boldsymbol{a}$ are uniformly distributed between 0 and $q-1$.

In Step 2, Alice and Bob sample the secret polynomials $\boldsymbol{s}$, $\boldsymbol{s'}$ and error polynomials $\boldsymbol{e}$, $\boldsymbol{e'}$ and $\boldsymbol{e''}$ from a binomial distribution χ. The sampling process requires, similar to the generation of $\boldsymbol{a}$, a lot of random bytes. Again a hash function (SHAKE-256) is used to expand a random seed. All resulting coefficients of these polynomials are relatively small and distributed between $-8 \mod q$ and $+8$.

In Step 3, the R-LWE instance $\boldsymbol{b}$ is created. Due to the hardness of recovering $\boldsymbol{s}$ from $\boldsymbol{b}$, $\boldsymbol{b}$ can be sent securely, together with the public seed, to Bob.

In Step 4, in case of a usual key exchange, a random key is assigned to the vector m. When NewHope is used for Public Key Encryption (PKE), the message that should be encrypted is assigned to m. The secret vector m is transformed to the secret polynomial $\boldsymbol{d}$ using the function NHEncode.

In Step 5, two further R-LWE instances, $\boldsymbol{u}$ and $\boldsymbol{c}$, are created. The R-LWE instance $\boldsymbol{c}$ is used to hide the secret polynomial $\boldsymbol{d}$. After receiving $\boldsymbol{c}$ and $\boldsymbol{u}$, Alice removes the largest noise terms from $\boldsymbol{d}$ by subtracting $\boldsymbol{us}$ from $\boldsymbol{c}$. After the decoding step, NHDecode, Alice and Bob share with a certain probability the same secret key or message m and m'.

B. Number Theoretic Transform (NTT)

The polynomial multiplications and hash generations are the performance bottleneck of NewHope. To reduce the complexity of the multiplication to $\mathcal{O}(n \log n)$, the NTT can be used because $\boldsymbol{c} = \text{NTT}^{-1}(\text{NTT}(\boldsymbol{a}) \circ \text{NTT}(\boldsymbol{b}))$, where $\circ$ denotes a coefficient-wise multiplication. The NTT is a Fast Fourier Transform (FFT) where the n-th primitive root of unity ω_n is an element of a finite field instead of the complex numbers. To transform the coefficients of polynomial $\boldsymbol{a}$ into the spectral and back into the normal domain, Eq. 1 and Eq. 2 can be used:

$$\hat{a}_i = \sum_{j=0}^{n-1} \gamma_n^j \cdot \omega_n^{ij} \cdot a_j \quad \text{and} \tag{1}$$

$$a_i = n^{-1} \cdot \gamma_n^{-i} \sum_{j=0}^{n-1} \omega_n^{-ij} \cdot \hat{a}_j \ , \tag{2}$$

where $n-1$ is the degree of the polynomial, ω_n the n-th root of unity, and $\gamma_n = \sqrt{\omega_n}$, which is used to avoid reductions $\mod (x^n + 1)$, is its square root.

The Cooley–Tukey technique exploits the symmetric structure of the NTT and breaks the equations into a sum with even and a sum with odd indices to increase efficiency. The basic algorithm is shown in Algorithm 1. Note that the algorithm requires a bit reversal of the input coefficients. The Twiddle factors (powers of ω_n) and the powers of γ are calculated during run-time to avoid a large number of pre-calculations and high memory usage. Instead of storing n Twiddle factors only $\log(n)$ values for $\omega_m = \omega_n^{n/m}$ are stored. Line 7 to 10 is the core of the algorithm, also denoted as butterfly operation.

Algorithm 1: NTT transform

```
   Input: Coefficients a_i, pre-calculated values of ω_m
   Result: Coefficients â_i
1  a ← BitReversal(a)
2  for m = 2 to n by m = 2m do
3      ω_m ← ω_n^{n/m},  ω ← 1
4      for j = 0 to m/2 − 1 by 1 do
5          for k = 0 to n − 1 by m do
6              t ← ω · a_{k+j+m/2}
7              u ← a_{k+j}
8              a_{k+j} ← u + t
9              a_{k+j+m/2} ← u − t
10         end
11         ω ← ω · ω_m
12     end
13 end
```

III. Related Works

Efficient software implementations of the NewHope algorithm were already proposed in [3] and [4]. The authors in [3] use Advanced Vector Extensions (AVX), which are supported by Intel processors since the Sandy Bridge generation, to significantly speed up their design. NewHope was also implemented in [4] on the popular ARM Cortex-M microcontroller family. The authors have shown that by careful optimization at assembler level it is possible to speed up critical operations.

However, in order to achieve more significant speed and energy improvements, the use of hardware accelerators is necessary. Previously proposed hardware implementations focused on the NTT transformation, which is used for speeding

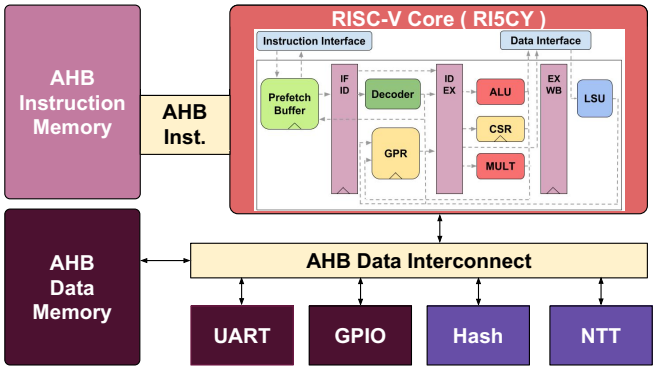

Fig. 1. Target RISC-V System-on-Chip Architecture

up large polynomial multiplications. Two different implementation approaches exist. The first approach, as in [5], [6], requires pre-calculations to store the Twiddle factors (powers of ω_n in Eq. 1) and other values together with the NTT input and output in the memory. The second approach, as in [7], [8], calculates the Twiddle factors during run-time to reduce the high memory overhead. The authors of [8] optimized the proposal of [7] by introducing a smart memory access scheme.

So far, none of the previous works considered a HW/SW co-design, which combines the advantages of both implementation methods and offers at the same time high flexibility and high performance.

IV. RISC-V Architecture

The target hardware platform used in this work is a minimalist System-on-Chip, showed in Fig. 1. The processing element is a RISC-V core from the Pulpino distribution [9], known as RI5CY core. The interconnection mechanism chosen was the AMBA AHB, which enables medium to high performance communication when handling a single master and few peripherals. There are two AHB buses instantiated in this design, one for the instruction port (AHB Inst.) and the other for the data port (AHB Data). The instruction port is only connected to an instruction memory. At the data port, the AHB integrates a data memory, an UART serial interface, a GPIO, a Timer, and our developed NTT and hash accelerators. This hardware platform was simulated at RTL, and synthesized on an FPGA. Since there is no boot loader, all software is hard-coded in the memories as initialization files; during synthesis, the instruction memory becomes a Read-Only Memory (ROM).

The RI5CY core is a four stage in-order pipeline 32-bit processor. Its main components, as showed in Fig. 1, are a prefetch buffer, an instruction decoder, a general purpose register bank (GPR), an arithmetical and logical unit (ALU), a multiplication unit (MULT), a control and status register unit (CSR), and a load-store unit (LSU). This core provides full support to the RV32I instruction set architecture, and partial support of the RV32M (only the multiplication instruction).

To develop applications and test the system operations, all elements were mapped to memory addresses. This organization was used to configure the AHB Data decoder, and also to write a software library, that works as a driver for applications usage. The memory map employed can be observed in Table I.

TABLE I
RISC-V SoC Memory Map

Address Range	Peripheral	Description
0x01000000 0x0100FFFF	Data Memory	Data memory.
0x1A100000 0x1A10FFFF	GPIO	GPIO pins (connected to LEDs).
0x1B100000 0x1B10FFFF	TIMER	Cycle accurate timer.
0x1C100000 0x1C10FFFF	UART	Interface to the HOST PC.
0x1D100000 0x1D10FFFF	NTT	Hardware NTT accelerator.
0x1E100000 0x1E10FFFF	Hash	Hardware hash accelerator.

V. Hardware Accelerator

To increase the efficiency of NewHope, two hardware accelerators were developed: the NTT and hash accelerator.

A. NTT Accelerator

The architecture of our NTT accelerator is shown in Fig. 2. The configuration signal *NTT_config* can be accessed by the memory address 0x1D100000. The accelerator has five different configuration modes: i) idle, ii) write data to accelerator memory, iii) read data from accelerator memory, iv) NTT operation, and v) NTT^{-1} operation.

During the write mode, the coefficients of a polynomial are written in bit reversed order into the RAM block. While software implementations require large LUTs or at least $\mathcal{O}(\log n)$ operations for the bit reversal, in hardware the wires of the address signal (AddrA) can be simply interchanged when writing the input data (DIA) to the memory. Our NTT accelerator uses the memory access scheme proposed in [8]. Each memory line stores two 16 bit coefficients that are processed together. At the beginning all even coefficients are written into the lower half-words and all odd coefficients into the higher half-words. After storing all coefficients, the accelerator can be either configured to perform the NTT or NTT^{-1} operation. Depending on the *NTT_config* signal either the small LUT for ω_m or ω_m^{-1} is selected. The *Address Unit* controls the memory read and write access required for the transformation. In each iteration of the Cooley–Tuckey algorithm, the lower and higher halfwords are loaded from the memory and assigned to L1 and H1, respectively. The *Butterfly Unit* performs the multiplications with the Twiddle factors, the additions and the subtractions. The multiplier of the *Butterfly Unit* is also reused for updating ω. To avoid preprocessing (multiplications with powers of γ in Eq. 1), ω can be initialized with $\gamma_m = \sqrt{\omega_m}$ as in [8]. In contrast to [8], we also integrate post-processing (multiplications with powers of γ and n^{-1}) into the NTT algorithm. The post processing step is only required during the last round of the NTT^{-1} operation.

B. Hash Accelerator

The hash accelerator is used to expand the random seed in order to generate the polynomials $\boldsymbol{a}$, $\boldsymbol{s}$, $\boldsymbol{s'}$, $\boldsymbol{e}$, $\boldsymbol{e'}$ and $\boldsymbol{e''}$.

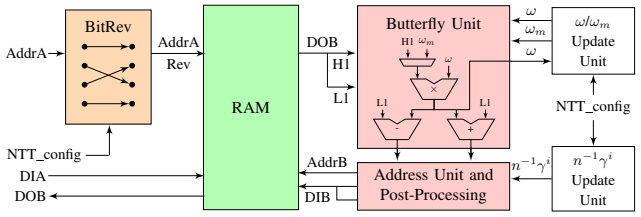

Fig. 2. NTT accelerator

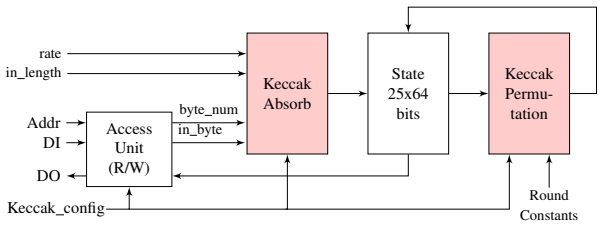

Fig. 3. Hash accelerator

More specifically, the extendable-output functions SHAKE-128 for generating the polynomial $\boldsymbol{a}$ and SHAKE-256 for generating the secret and error polynomials are used. The SHAKE functions are part of the SHA-3 hash family, which was originally developed under the name Keccak.

Figure 3 shows our hash hardware accelerator. The hash generator has two main modules: the *Keccak Absorb* and the *Keccak Permutation* module. The *Keccak_config* signal, accessible at address 0x1E100000, triggers the *Keccak Absorb* and *Keccak Permutation* module and configures the read/write access module. The *Keccak Absorb* module takes as input the true random seed and the parameters *rate* and *in_len* (input length). The parameter *rate*, which determines the output bit length, and *in_len* can be accessed at address 0x1E100004 and 0x1E100008, respectively. During the absorption phase the input is transformed into a state with 25x64 bits. In the next phase, the *Keccak Permutation* module, which is based on the implementation in [10], transforms the state using the Keccak round function, consisting of shift, XOR, AND and NOT operations. After the permutation phase, the output can be taken from the state register. The output hash length depends on the rate and is set to 1344 bit for SHAKE-128 and 1088 bit for SHAKE-256. After taking the 1344 or 1088 bit, the permutation phase can be repeated until the number of desired output bits is obtained. As SHAKE has no predefined output length, it serves as an ideal pseudo random number generator.

VI. Fault Analysis

Our fault analysis is divided into two parts. First, we analyze the reliability of the NTT computation with respect to random faults and then we propose a fault attack model and countermeasures.

A. Resilience Evaluation

We carry out simulation-based *fault-injection* (FI) campaigns [11] to assess the resilience of both NTT implementation variants (with pre-calculated LUTs for Twiddle factors, powers of γ and bit reversal, and without). Pure SW-based implementations were executed on our existing *ETISS-ML* [12] simulator. The tool enables the simulation of the impact of *soft-errors* [13] by injecting *bit-flips* randomly in space (CPU flipflop) and time (clock-cycle) into a RTL model of the processor, while it executes the NTT routine. We perform 10,000 such experiments for each of the two NTT variants. The impact of each soft error is classified as follows:

- *Micro-architecture Masking (μAM):* The fault is masked within the micro-architecture of the processor before impacting a software-visible state such as the register file or memory.
- *Application Masking (AM):* The fault propagates into a software visible state, however the outputs of the NTT computation is not corrupted.
- *Unexpected Termination (UT):* The execution ends up in an exception-handler, as a result of a CPU exception.
- *Hang (H):* The CPU crashes and stops execution.
- *Silent Data Corruption (SDC):* The output of the NTT is corrupted and no exception is reported by the processor.
- *Latent (LAT):* The fault remains in the processor but does not propagate.

Table II (first two rows) show the corresponding outcome rates for the variants NTT-LUT and NTT. The soft error resilience or failure rate is usually set to be the SDC rate as these are corrupted outputs that are not reported by the processor. Precisely, the effective failure-rates (SDC) for both NTT-LUT and NTT are shown to be 4.96% and 3.66%. Hence, the NTT approach has slightly better soft error resilience.

TABLE II
Soft error Classification

	μAM	AM	UT	H	SDC	LAT
NTT-LUT	59.92%	25.58%	2.94%	0.98%	4.96%	5.62%
NTT	59.58%	26.38%	2.78%	0.92%	3.66%	6.68%
NTT-LUT	58.72%	29.02%	2.98%	1.06%	2.14%	6.08%
DMR Det.	0%	7.99%	2.68%	0%	83.17%	0%
NTT	60.12%	28.12%	2.68%	0.96%	2.08%	6.04%
DMR Det.	0%	6.54%	0.74%	0%	80.77%	0%

Execution redundancy is a generally applicable technique to reduce the effective SDC of a given SW application [14]. We harden both NTT implementations using Dual Modular Redundancy (DMR) principles. The corresponding results are shown in Table II. The DMR detection rates are also expressed in terms of percentage. The DMR was able to detect 168 out of 208 SDC scenarios thus resulting in a SDC detection rate of 80.77%. The effective SDC rates for NTT-LUT and NTT drop down to 0.72% and 0.8% respectively. Furthermore, these redundancy-based techniques do not suffer from inherent performance overhead when targeted towards parallel architectures, hence further motivating the case for a HW/SW co-design implementation approach.

B. Fault Attack Model

Besides increasing resilience against random faults, it is also crucial to consider malicious fault attacks. Fault attacks

intentionally manipulate a cryptographic system at the physical level to leak information about secret parameters. Typical fault injection methods are power supply manipulations, clock glitching, temperature attacks, optical attacks and electromagnetic attacks. In particular, optical attacks have a high localization and timing precision. They use a strong light source, e.g. a laser, to cause an ionization effect in the transistors and can even set or reset a single bit [15].

A fault attack on NewHope was already proposed in [16]. The authors manipulated the seed that is passed to the sampling functions. NewHope concatenates the same seed with a nonce (number that is used only once) to sample the secret and error polynomials. The authors manipulated this nonce so that the sampling routine does not output anymore different secret and error polynomials, introducing a severe security issue.

In the following, we propose another attack scenario on the NTT^{-1} operation. Let us assume an attacker who is able to temporarily reset a specific word in the data memory or register of the RISC-V core. The main secret elements in Protocol 1 are the secret polynomial $\boldsymbol{s}$ and the key/message m. The aim of our attack is to reveal the message m in a PKE setting. During the encoding NHEncode (Step 4 Protocol 1) one bit of the vector m is encoded into four coefficients of the polynomial d. Depending on whether the respective message bit is zero or one, the coefficients are encoded to either zero or $\lfloor q/2 \rfloor$, respectively. The polynomial $\boldsymbol{d}$ is now hidden in polynomial $\boldsymbol{c} = \boldsymbol{bs'} + \boldsymbol{e''} + \boldsymbol{d}$. The polynomial multiplication of $\boldsymbol{bs'}$ is realized with NTT^{-1}(NTT($\boldsymbol{b}$) $\circ$ NTT($\boldsymbol{s'}$)). Now, when the output coefficients of the NTT^{-1} are forced to zero, the ciphertext is equal to $\boldsymbol{c} = \boldsymbol{e''} + \boldsymbol{d}$. As the coefficients of $\boldsymbol{e''}$ are very small, the ciphertext $\boldsymbol{c} \approx \boldsymbol{d}$. Then, the manipulated ciphertext is sent to Bob over the insecure public channel, which can be eavesdropped by the attacker. The output coefficients of the NTT^{-1} are multiplied with n^{-1} and γ^{-i} (Eq. 2). If we set one of these parameters to zero, the resulting coefficient is zero, too.

The register or memory location that has to be attacked highly depends on the implementation. In our software implementation, a variable γ is initialized with n^{-1} and multiplied with γ^{-1} in each iteration of the last inner loop. The constants n^{-1} and γ^{-1} are defined with a macro in the C header file. As the macro is only a text substitution, the constants are not loaded into the data memory. However, they are loaded during the execution of the NTT^{-1} into the registers R19 and R29, respectively. If an attacker sets one of these constants to zero, the value zero propagates and forces all subsequent coefficients to zero. In the NTT hardware implementation a simple Hardware Trojan in form of a multiplexer could be employed that forwards in the attacking mode the value zero instead of $n^{-1}\gamma^{-i}$ to the post-processing unit.

The DMR, which was used to decrease the SDC rate, is also an effective method for detecting fault attacks. Another simple countermeasure is to check if the last coefficient of the output polynomial is zero, which is very unlikely at normal operation. If it is, the transmission to Bob is rejected. But this countermeasure would not prevent different fault attacks.

VII. Test Results

Our SoC platform was implemented on the Zedboard, which is equipped with a Xilinx Zynq-7000. First, we ported the NewHope-NIST Chosen-Plaintext Attack (CPA) secure reference implementation to our RISC-V platform. We used the NewHope parameter set $n = 1024$ (dimension) and $q = 12289$ (modulo reduction parameter), which results into the highest NIST security level. The SW reference implementation was then optimized with the NTT and hash accelerators.

The cycle counts of our test results and related works are provided in Table III and the resource utilization of the hardware implementations in Table IV. The reference implementation in [3] uses an Intel Core i7-4770K (Haswell) CPU to measure the amount of clock cycles. They also used Single Instruction Multiple Data (SIMD) instructions of the AVX2 instructions set to optimize the performance. Clearly, this high performance platform results into a very low number of clock cycles. In [4], the authors optimized NewHope on assembly level and ported it to the microcontroller platforms Cortex-M0 and Cortex-M4. Their work has shown that with careful optimizations on assembler level a high improvement can be achieved. Our RISC-V SW implementation is based on the reference C code in [3]. In contrast to the other software implementations, we avoided at the NTT computation large LUTs for the bit reversal, Twiddle factors and the powers of γ. Our software was compiled with the "GNU MCU Eclipse RISC-V Embedded GCC Version 7.2.0" toolchain without using optimization flags. Note that RISC-V is a very new technology and the compiler is not yet highly developed.

In [8], the authors propose a good trade-off between performance and memory usage for the NTT transformation. The authors implemented the basic R-LWE scheme without key generation phase for a smaller dimension of $n = 512$. Therefore, it is difficult to directly compare it with the other implementations. The authors in [6] have a very fast implementation for NewHope-USENIX but a high BRAM utilization. In [5], the direct predecessor of NewHope-NIST (NewHope Simple) was implemented. They optimized for area while keeping a decent performance.

Our NTT implementation requires only one BRAM for storing the coefficients. The $\log(n)$ values for ω_m and ω_m^{-1} can be stored in a small LUT. The high number of DSP slices comes from the constant time modulo multiplier, modulo adder and modulo subtractor, which are based on the Montgomery and Barrett algorithms. Constant time modulo reduction is necessary to avoid timing attacks, such as in [17]. Writing the coefficients from the RISC-V core to the memory and reading the transformed coefficients requires 7,178 clock cycles, respectively. The absorbing phase of our hash accelerator requires one clock cycle and the 24 rounds for the permutation phase require 12 clock cycles. The total amount of clock cycles for generating $\boldsymbol{a}$ and sampling the secret and error polynomials is composed by the time for the transfer of input/output data, the absorption phase, the repetitive calls of the permutation module and the creation

TABLE III
CYCLE COUNT NEWHOPE/R-LWE IMPLEMENTATIONS

		Gen_a	Sample	NTT	NTT^{-1}	Key gen.	Encaps.	Decaps.	Total
SW	**Ref. Intel AVX [3]**	32 248	–	49 920[a)]	53 596[a)]	222 922	330 828	87 080	640 830
	Opt. Intel AVX [3]	21 308	–	8 416[a)]	11 708[a)]	107 032	163 332	35 716	306 080
	Cortex-M0 [4]	328 789	208 692[b)]	148 517[a)]	167 405[a)]	1 170 892	1 760 837	298 877	3 230 606
	Cortex-M4 [4]	263 089	111 794	86 769[a)]	97 340[a)]	781 518	1 140 594	174 798	2 096 910
	This work SW	1 667 554	1 708 426	344 403	450 813	5 990 806	8 154 231	611 235	14 756 272
HW	**Roy *et al.* [8]**	–	–	3 443[c)]	4 775[c)]	–	13 300	5 800	19 100
	Kuo *et al.* [6]	–	–	–	–	8 600	11 300	2 800	22 700
	Oder *et al.* [5]	–	33 794	35 845[a)]	45 064[a)]	115 784	179 292	55 340	350 416
	This work HW/SW	42 050	75 682	24 609	24 609	357 052	589 285	167 647	1 113 984

[a)] With LUTs for Twiddle factors, powers of γ and bit reversal for SW implementations.
[b)] With ChaCha20 instead of SHAKE functions [c)] For polynomial degree $n = 512$

TABLE IV
RESOURCE UTILIZATION NEWHOPE/R-LWE

	LUTs	FFs	BRAMs	DSPs
Roy *et al.* [8]	1 536	953	3	1
Kuo *et al.* [6]	12 340	6 098	14	29
Oder *et al.* [5]	9 640	9 087	8	4
This work:				
– RISC core	6 453	2 184	0	6
– Peripherals/Memory	8 832	276	0	0
– NTT accelerator	886	618	1	26
– Hash accelerator	10 435	4225	0	0

of the polynomial from the expanded seed. Due to increasing communication overhead, our HW/SW co-design is slower than the best pure hardware implementations but outperforms existing microcontroller implementations. The NTT and hash hardware accelerators bring a huge performance boost. At the same time our HW/SW design offers high flexibility. The algorithm, security level, and hardening against random noise as well as fault attacks can be easily changed or integrated on software level.

VIII. CONCLUSION

In this work, we proposed a co-processor for lattice-based cryptography. As use-case, we implemented the prominent cryptographic scheme NewHope on a RISC-V based SoC platform. Our test results have shown that, even with increasing data transfer, a significant speed up can be achieved by using an NTT and hash hardware accelerator. Moreover, we investigated the reliability of NTT software implementations regarding random fault injections. Not only random faults but also precise injections by an attacker are of major concern. We developed a realistic fault attack scenario and propose simple countermeasures to harden the design against random and adversarial fault injections.

Acknowledgments. This work was partly funded by the German Ministry of Education, Research and Technology in the project ALESSIO through the grant number 16KIS0632.

REFERENCES

[1] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

[2] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange-a new hope." in *USENIX Security Symposium*, vol. 2016, 2016.

[3] E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, T. Pöppelmann, P. Schwabe, and D. Stebila, "NewHope: Algorithm Specifcations and Supporting Documentation," 2017.

[4] E. Alkim, P. Jakubeit, and P. Schwabe, "Newhope on arm cortex-m," in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2016, pp. 332–349.

[5] T. Oder and T. Güneysu, "Implementing the newhope-simple key exchange on low-cost fpgas," *Progress in Cryptology–LATINCRYPT*, vol. 2017, 2017.

[6] P.-C. Kuo, W.-D. Li, Y.-W. Chen, Y.-C. Hsu, B.-Y. Peng, C.-M. Cheng, and B.-Y. Yang, "Post-quantum key exchange on FPGAs," *IACR ePrint*, vol. 690, p. 2017, 2017.

[7] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient FPGA implementations of lattice-based cryptography," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, 2013, pp. 81–86.

[8] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014, pp. 371–391.

[9] A. Traber, S. Stucki, F. Zaruba, M. Gautschi, A. Pullini, and L. Benini, "Pulpino: A risc-v based single-core system," 2015, openRISC Conference, ORCONF2015; Conference Location: Geneva, Switzerland; Conference Date: October 9-11, 2015; .

[10] H. Hsing. (2012) OpenCores. SHA3 (KECCAK). [Online]. Available: https://opencores.org/projects/sha3

[11] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A survey on fault injection techniques," pp. 171–186, 2004.

[12] D. Mueller-Gritschneder, U. Sharif, and U. Schlichtmann, "Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator etiss-ml," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 127:1–127:8.

[13] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[14] L. Osinski, T. Langer, and J. Mottok, "A survey of fault tolerance approaches on different architecture levels," in *ARCS 2017; International Conference on Architecture of Computing Systems*, 2017, pp. 1–9.

[15] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2295–2306, 2013.

[16] P. Ravi, S. Bhasin, and A. Chattopadhyay, "Number "not" used once - key recovery fault attacks on LWE based lattice cryptographic schemes," *IACR Cryptology ePrint Archive*, vol. 2018, p. 211, 2018.

[17] R. Primas, P. Pessl, and S. Mangard, "Single-trace side-channel attacks on masked lattice-based encryption," in *CHES*, ser. Lecture Notes in Computer Science, vol. 10529. Springer, 2017, pp. 513–533.