Towards CRYSTALS-Kyber VHDL Implementation

Sara Ricci[a], Petr Jedlicka[b], Peter Cibik[c], Petr Dzurenda[d], Lukas Malina[e] and Jan Hajny[f]
Department of Telecommunications, Brno University of Technology, Brno, Czech Republic

Keywords: Post-quantum Cryptography, Lattice-based Cryptography, Key Encapsulation Scheme, Number-theoretic Transform, FPGA, VHDL Implementation, Parallelization, Optimization.

Abstract: Kyber is one of the three finalists of the National Institute of Standards and Technology (NIST) post-quantum cryptography competition. This article presents an optimized Very High Speed Integrated Circuit Hardware Description Language (VHDL)-based implementation of the main components of the Kyber scheme, namely Number-Theoretic Transform (NTT) and Keccak. We focus specifically on NTT, Keccak and their derivatives since they largely determine Kyber's performance due to their wide involvement in each step of the scheme. Our high-speed implementation also takes into account the trade-off between the degree of parallelization and the resources utilization. The NTT component is more than 27% faster than the state-of-the-art implementations. Furthermore, the optimization helps the algorithm to achieve 1 572 839 NTT operations per second.

1 INTRODUCTION

In 2016, the NIST initiated a call for proposal of new Post-Quantum Cryptography (PQC) digital signatures and Key Encapsulation Mechanisms (KEMs) for future standardization (NIST, 2016). In January 2019, NIST announced the third round finalists, where 3 signatures and 4 KEMs were selected as potential future standards from the competing 69 candidates (NIST, 2019). These schemes were chosen based on their security strength, and software and hardware performance.

While most of the finalists already include an implementation optimized for large x86 processors, they often lack optimized implementations for other platforms. Table 1 shows the current state of NIST PQC finalists VHDL-based and High-Level Synthesis (HLS)-based implementations on the Field Programmable Gate Arrays (FPGA) platform (references are given in the table). All schemes have HLS-based implementations which have been published in different articles. However, the outputs from HLS are often less efficient than native VHDL-based implementations, and could present security bugs. As shown in Table 1, no pure VHDL implementation exists so far.

Remarkable is that 3 out of 4 third round finalists for KEM are based on structured lattices, i.e., they rely on arithmetic in polynomial rings. Indeed, Lattice-based cryptography has gained significant attention for its performance among the PQC families. Especially CRYSTALS-Kyber (Avanzi et al., 2017), shortly Kyber, which is designed to support very efficient multiplication over polynomial ring without additional memory. It is important to notice that Kyber is identified as the fastest KEM after pipelining in the PQC NIST KEM semifinalists comparison done in (Basu et al., 2019).

1.1 Related Work

Several partial or whole FPGA implementations of Kyber are currently available. The used technologies can be split in three groups: HLS-based design (e.g., ANSI C/C++, and Matlab), software-hardware co-design and RTL-based design (e.g., VHDL, and Verilog).

(Chen et al., 2021) propose a polynomial ring processor for Kyber on a Xilinx Artix-7 series FPGA platform using HLS technology. They develop an optimized NTT which uses convolution-based polynomial multiplier. (Basu et al., 2019) employ the HLS method to implement and compare 11 NIST PQC semifinalists on the Xilinx Virtex-7 FPGA platform. CRYSTALS-Kyber is recognized as the fastest KEM

[a] https://orcid.org/0000-0003-0842-4951
[b] https://orcid.org/0000-0003-0833-8068
[c] https://orcid.org/0000-0003-0780-6288
[d] https://orcid.org/0000-0002-4366-3950
[e] https://orcid.org/0000-0002-7208-2514
[f] https://orcid.org/0000-0003-2831-1073

Ricci, S., Jedlicka, P., Cibik, P., Dzurenda, P., Malina, L. and Hajny, J.
Towards CRYSTALS-Kyber VHDL Implementation.
DOI: 10.5220/0010580407600765
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 760-765
ISBN: 978-989-758-524-1

Table 1: VHDL and HLS implementations of NIST PQC finalists.

Digital Signature			
Scheme	Type	HLS method	pure VHDL implementation
Dilithium	lattice	(Soni et al., 2019; Soni et al., 2020) (Basu et al., 2019)	(Ricci et al., 2021)
Falcon	lattice	(Soni et al., 2020)	✗
Rainbow	multivariate	(Soni et al., 2020)	(Ferozpuri and Gaj, 2018)
Encryption/KEM			
Scheme	Type	HLS method	pure VHDL implementation
Kyber	lattice	(Basu et al., 2019; Chen et al., 2021)	✗
McEliece	code	(Soni et al., 2019; Basu et al., 2019)	(Wang et al., 2018)
NTRU	lattice	(Soni et al., 2019; Basu et al., 2019)	(Marotzke, 2020)
SABER	lattice	(Soni et al., 2019; Basu et al., 2019)	(Roy and Basso, 2020)

Note: ✗– pure hardware implementations without language specification, ✗– algorithm is not implemented.

under pipelining directives.

(Dang et al., 2020) present a software-hardware co-design approach for the implementation of three NIST semifinalists: Kyber, NewHope and Round5 schemes. They combine C code with Register-Transfer Level (RTL) design methodology on the Xilinx Artix-7 FPGA family.

Even if some Kyber RTL-based designs are now available, there is still a lack of its optimized implementation on FPGA. A Verilog Kyber implementation on Xilinx Artix-7 and Virtex-7 FPGAs is presented in (Huang et al., 2020). The article stays on a high level description of Kyber and no NTT and Keccak results are shown. Furthermore, (Chen et al., 2020) design an NTT optimization with Gentlemen-Sande butterfly on Xilinx XC7A200T and XC6SLX45T FPGA platforms. However, no specification on the used language are given. At last, (Xing and Li, 2021) develop a Kyber hardware implementation on the Xilinx XC7A12TCPG238-1 FPGA platform. But even here, no specification on the used language are given. Moreover, this lightweight platform has small memory capacity and all the implementation has been design to run on this characteristic. On the contrary, our implementation works on the high-end Xilinx Virtex UltraScale+ XCVU7P FPGA platform. Due to the bigger available resources, we could focus on a high-speed implementation, i.e. on scheme performance and optimization.

1.2 Contributions

The main contribution of this paper is to present optimized NTT and Keccak VHDL implementations for Kyber on high-speed FPGA platform. Our high-speed platform implementation targets the minimization of the execution times of the major operations by components optimization and operations/sub-components parallelization. Moreover, our implementation focuses on the trade-off between the degree of parallelization and the resources utilization. Our NTT is more than a factor of 25 and 3 times faster than the performance presented in (Chen et al., 2020) and (Dang et al., 2020), respectively. Our Keccak implementation is easy to use in specific instances of SHA and SHAKE functions. In fact, each derivative has its specific implementation of Keccak core blocks `hash_core` and `pad10*1`.

2 PRELIMINARIES

In this section, we discuss the mathematical background that is crucial for the understanding our implementation. Moreover, we recall NTT, Keccak and Kyber scheme. Finally, we describe the main characteristic of FPGA implementations and how they can be compared. We denote by $\mathcal{R}_q$ the polynomial ring $\mathbb{Z}[x]_q/(x^n+1)$ where $n = 256$ and $q = 3329$. Bold lower-case letters ($\mathbf{v}$) are used for column vectors in $\mathcal{R}_q$, while regular font letters (v) for elements in $\mathcal{R}_q$. Matrices are represented by bold upper-case letters ($\mathbf{A}$). The coefficient-wise multiplication of two polynomials in the NTT domain with the natural extension to vectors and matrices is denoted by $\circ$.

NTT. NTT is a very efficient way to perform multiplications in polynomial rings (Nejatollahi et al., 2019). In NTT, a polynomial becomes a multi-point evaluation at powers of a root of unity. Therefore, the polynomial multiplication consists in applying NTT in $O(n\log n)$, then performing point-wise multiplication in $O(n)$ and finally converting the result to a coefficient representation in $O(n\log n)$. There are many ways to compute the number-theoretic transform. Ky-

ber uses Cooley-Tukey butterfly for NTT, Gentleman-Sande butterfly for NTT^{-1}, Montgomery algorithm for modular multiplication and Barrett algorithm for modular reduction.

Keccak. The performance of Kyber is largely determined by hashing and pseudorandom number generation. All these symmetric primitives require Keccak permutation (Bertoni et al., 2009). Keccak is a family of sponge functions using Keccak-f permutation as the underlying function. The function f maps a single fixed-length string $b = r + c$, where r is the bitrate and c the capacity, to another string of the same length. It is important to mention that the Kyber scheme includes several hash functions such as `SHA3-256` and `SHA3-512`, and extendable output functions `SHAKE-128` and `SHAKE-256` which are all based on Keccak.

Algorithm 1: CPA KeyGen().

1: $\rho, \sigma \leftarrow \{0,1\}^{256} \times \{0,1\}^{256}$ ▷ `SHA3-512`
2: $\mathbf{A} \in R_q^{k \times k}$ ▷ `SHAKE-128` & NTT
3: $\mathbf{s}, \mathbf{e} \in R_q^k$ ▷ `SHAKE-256`
4: $\hat{\mathbf{t}} := \mathbf{A} \circ NTT(\mathbf{s}) + NTT(\mathbf{e})$ ▷ NTT
5: **return** $pk = (\rho, \hat{\mathbf{t}})$, $sk = \mathbf{s}$

Kyber. CRYSTALS-Kyber scheme (Avanzi et al., 2017) is part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), which counts KEM, namely Kyber, and a signature, namely Dilithium. Both protocols' security relies on the hardness of the Module variant of the Learning With Error (MLWE) problem (Brakerski et al., 2014; Langlois and Stehlé, 2015).

Algorithm 2: CPA Encryption($\rho, \hat{\mathbf{t}}, m$).

1: $\hat{\mathbf{A}} \in R_q^{k \times k}$ ▷ `SHAKE-128` & NTT
2: $\mathbf{r}, \mathbf{e}_1 \in R_q^k$ ▷ `SHAKE-256`
3: $e_2 \in R_q$ ▷ `SHAKE-256`
4: $\hat{\mathbf{r}} := NTT(\mathbf{r})$ ▷ NTT
5: $\mathbf{u} := NTT^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + NTT(\mathbf{e}_1)$ ▷ NTT
6: $v := NTT^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + m$ ▷ NTT
7: **return** $c = (\text{Compress}(\mathbf{u}), \text{Compress}(\mathbf{v}))$

Kyber uses the MLWE problem with n and q fixed while the dimension of the module is being varied. This gives the advantage that changing the security level of Kyber involves doing more (or fewer) of the same ring operations. On the contrary, changing the ring would require the re-implementation of all the operations. Kyber is constructed in two-stage approach: 1) Basic version `Kyber.CPAPKE` and 2) extended version `Kyber.CCAKEM` (shortly Kyber), see (Avanzi et al., 2017) for more information.

Algorithm 3: CPA Decryption($sk = \mathbf{s}, c$).

1: $\mathbf{u} := \text{Decompress}(c_1)$
2: $v := \text{Decompress}(c_2)$
3: $\mathbf{A} \in R_q^{k \times k}$
4: $m :=$Compress$(v - NTT^{-1}(\mathbf{s}^T \circ NTT(\mathbf{u})))$ ▷ NTT
5: **return** m

Algorithms 1, 2 and 3 show a brief description of the main core of `CPAPKE.Kyber` scheme. In the aforementioned algorithms, NTT and Keccak derivatives usage are specified. `CCAKEM.Kyber` scheme variant differs from `CPAPKE.Kyber` on the usage of a hashed message m and a hashed public key pk in the encryption and decryption phases. In particular, `SHA3-256` is used for hashing. It is remarkable that the high presence of NTT and Keccak derivatives in all algorithms, in particular in Algorithms 1 and 2. In fact, these functions largely determine Kyber performance.

FPGA Implementations. Hardware implementations outperform software ones considering at least one of the following metrics: speed, power consumption, or energy usage (Dang et al., 2020). The FPGA implementations can be split into high-speed and lightweight. They try to minimize the execution times of the major operations by protocol optimization and operations/sub-components parallelization. On the contrary, lightweight implementations try to achieve minimum resource utilization, assuming of not exceeding a predefined maximum execution time. In particular, resource utilization is a vector counting the number of Logic Cells, Look-Up Tables (LUTs), Flip-Flops (FFs), Digital Signal Processor (DSP) slices and Block Random Access Memories (BRAMs).

3 VHDL IMPLEMENTATION

NTT Design and Implementation. The NTT and NTT^{-1} components consist of several functional sub-blocks including two butterflies, Montgomery algorithm, and Barrett algorithm. These sub-blocks are implemented and then integrated into the two components.

Whereas our design is speed-optimized, pipelined processing is applied and Digital Signal Processing (DSP) blocks are used for the implementation of all arithmetical operations necessary for NTT and NTT^{-1} functions. Moreover, the arithmetical oper-

Table 2: Hardware specifications of the FPGA platforms for Kyber components.

Design	Platform	Logic Cells / LUTs	FFs	DSP	BRAMs / Block Size
High-speed platforms					
Our work	**UltraScale+ XCVU7P**	**1 724 100 / 788 160**	**1 576 320**	**4 560**	**1 440 / 36 Kb**
(Dang et al., 2020)	UltraScale+ Zynq XCZU9EG	599 550 / 274 080	548 160	2 520	912 / 36 Kb
(Huang et al., 2020)	Virtex-7 VC707 XC7VX485T	485 760 / 303 600	607 200	2 800	1 030 / 36 Kb
Lightweight platforms					
(Chen et al., 2021)	Artix-7 XC7A200T	215 360 / 134 600	269 200	740	365 / 36 Kb
(Xing and Li, 2021)	Artix-7 XA7A12T	12 800 / 8 000	16 000	40	20 / 36 Kb
(Chen et al., 2020)	Spartan-6 XC6SLX45T	43 661 / 27 288	54 576	58	116 / 18 Kb

ations are run in parallel with a throughput equal to one output per one clock cycle.

The computation of NTT or NTT^{-1} functions is performed over 256 coefficients in 7 iterations. As trade-off between the degree of parallelization and hardware resources, 4 butterflies in a 2x2 arrangement are deployed. This leads to the computation of 2 NTTs iterations in parallel with 2 butterflies in each of them. It is remarkable that the 7th iteration is performed using only two butterflies in the first layer while the remaining butterflies are bypassed. The butterflies are distributed in two iterations to reduce the number of coefficients or intermediate results needed to be stored in the Block Random Access Memory (BRAM) at one clock cycle. The block diagram of the NTT component is shown in Figure 1. The coefficients, intermediate results and output values are stored in BRAMs. The roots of unity needed for the NTT computation are stored in the Read-Only Memory (ROM) whose address is controlled by the control unit and data bus is accessed by the butterflies.

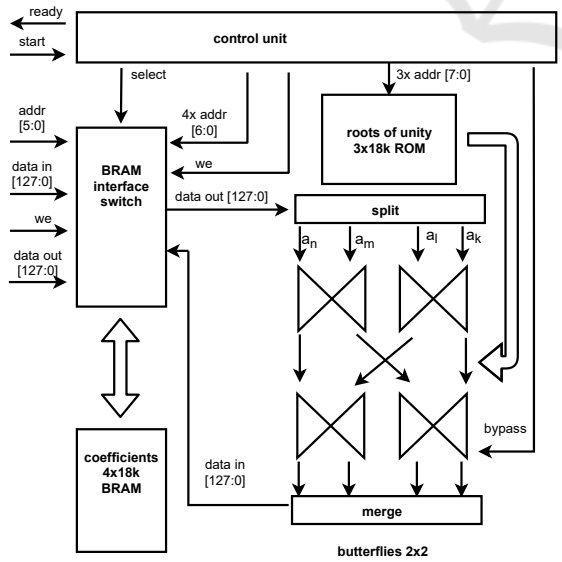

Figure 1: The block scheme of the NTT implementation.

The only difference between NTT^{-1} and NTT implementation is order in which the iterations are performed. In fact, in NTT^{-1} the distribution of the butterflies is reversed with respect to NTT component. The Montgomery reduction of the output coefficients is additionally performed for NTT^{-1} transformation.

Keccak and Derivatives Design and Implementation. Our Keccak implementation consists of two main components: `hash-core` and `pad10*1`. The Keccak sponge function is implemented in the `hash-core` component with parameter $b = 1600$ and counts 24 rounds. Our implementation computes two rounds per clock cycle, which means that the output value is pre-computed for 512-bit input approximately each 12 clock cycles. The `pad10*1` component is responsible of adding the multi-rate padding (`10*1`) and the specific derivative functions padding, e.g. "`1111`" and "`01`", to the last block of input message (NIST, 2015).

The derivative functions `SHA3-512`, `SHA3-256`, `SHAKE128` and `SHAKE256` are implemented as a top component with specific setting of internal subcomponents. Each top component has 512-bit input interface and for both input and output interfaces has handshake signals (`src` and `dst` ready) for transaction transmission.

4 EXPERIMENTAL RESULTS AND COMPARISON

This section describes the experimental results of our hardware implementation design for NTT, NTT^{-1}, Keccak function and Keccak derivatives used in the Kyber scheme. Moreover, we compare our work with existing implementations. Our implementation works with high-speed FPGA Xilinx Virtex UltraScale+ XCVU7P with manufacturer product number `xcvu7p-flvb2104-2-i`. Table 2 lists the basic hardware specifications of our FPGA and other FPGA platforms used in related works that are mentioned in this paper. The table categorizes FPGA platforms to high-speed and lightweight.

Table 3: Comparison of hardware resources and performance of NTT component of Kyber scheme on FPGA. "Max. Freq." states for maximum frequency, and "Ops" for operations per second. Not available parameters are marked as "-".

Design	LUTs	FFs	DSP	BRAMs	Latency Cycles	Max. Freq. [MHz]	Ops
Our Work	1 107	1 407	28	3.5	405	637	1 572 839
(Chen et al., 2020)[a]	442	237	1	1.5	2 055	136	66 180
(Chen et al., 2020)[b]	446	237	1	2	2 055	85	41 443
(Chen et al., 2021)	479	472	1	2	-	240	-
(Dang et al., 2020)[c]	2 325	2 346	24	7	1 271	455	357 986
(Dang et al., 2020)[d]	2 040	3 223	24	5	1 271	500	393 391
(Xing and Li, 2021)	1 737	1 167	2	3	-	-	-

Note: [a] with Artix7 FPGA platform, [b] with Spartan6 FPGA platform, [c] results of HLS implementation, [d] results of RTL implementation.

4.1 Performance Results

Tables 3 and 4 show our results after the synthesis of NTT components and Keccak component, respectively. The tables show the resources utilization (e.g., LUTs, FFs, DSP slices and BRAMs) and also the theoretical operating frequency and latency that can serve for the assessment of computational performance. Similarly to NTT, NTT^{-1} requires in average 1 572 LUTs, 2 002 FFs, 60 DSP, 3.5 BRAMs, 418 `cycles` Latency, and 637 `MHz` Frequency. In Keccak evaluation, DSP, and BRAMs are zero since they are not needed in the implementations. Note that NTT and NTT^{-1} require DSP and BRAMs due to their higher complexity. In particular, DSP slices are used for optimization reasons while BRAMs blocks are needed to store NTT (and NTT^{-1}) input, intermediate and output values.

Table 4: The hardware resources utilization of Keccak and derivatives components after an RTL synthesis. "Max. Freq." states for maximum frequency and "r" for bitrate.

Component	r [bit]	LUTs	FFs	Max. Freq. [MHz]
SHA3–512	576	18 543	4 829	237
SHA3–256	1 088	13 879	5 608	242
SHAKE128	1 344	18 448	8 873	230
SHAKE256	1 088	15 704	7 592	241

4.2 Comparison

Table 3 shows the comparison of several NTT implementations results. (Chen et al., 2020) present hardware NTT optimization with Gentlemen-Sande butterfly on Xilinx XC7A200T and XC6SLX45T FPGA platforms. However, no specifications on the used language are given. Checking the number of Operations Per Second (ops), their implementations are able to perform around 25 times less NTT transformations per second while using 4 times less hardware resources (LUTs and FFs). Moreover, if we define the *comparison effectiveness* of two implementations on two different FPGA platforms as (ops ratio)/(utilization ratio)/(theoretical max. frequency ratio), then our implementation is around 3.7 times more effective than (Chen et al., 2020). In particular, we assume that the typical theoretical maximum frequency of a low cost FPGA such as Artix7 is 464 MHz while our UltraScale+ is able to run at 775 MHz.

The differences between our and their implementation is due to the available resources of the used FPGA platforms. Our high-speed implementation also takes into account the trade-off between the degree of parallelization and the resources utilization. In fact, if we consider the most lightweight FPGA platform Spartan-6 XC6SLX45T used by (Chen et al., 2020), we need only ca. 4% LUTs, 3% FFs, 48% DSP, and 6% BRAMs of its available hardware resources. Furthermore, (Dang et al., 2020) hardware-software co-design reaches more than 3 times less NTT transformations per second while using twice more hardware resources. Regardless the platform they used, our implementation is more effective. In (Xing and Li, 2021) and (Chen et al., 2021), no NTT component performance results are provided.

Our Keccak implementation focuses mainly on performance. However, the impact on resource consumption remains negligible and comparable with other state-of-the-art solutions. In (Xing and Li, 2021), no performance results of Keccak and its derivatives (i.e., latency, maximum frequency, ops values) are provided, and therefore it is also hard to objectively compare our solutions.

5 CONCLUSIONS

In this paper, we present a pure and optimized implementation for the main algorithms of the Kyber scheme, namely The NTT and Keccak algorithms. The implementations are written in the VHDL language and performed on the UltraScale+ XCVU7P

FPGA chip. Our high-speed implementation also takes into account the trade-off between the degree of parallelization and the resources utilization. Our implementation is more efficient than currently existing implementations. In particular, our NTT implementation is ca 27% faster than (Dang et al., 2020) while using significantly less resources, and ca 155% faster than (Chen et al., 2021) proposal. Indeed, (Chen et al., 2021) is most resources friendly implementation. Note that if we consider the most lightweight FPGA platform Spartan-6 XC6SLX45T used by (Chen et al., 2020), we need only ca. 4% LUTs, 3% FFs, 48% DSP, and 6% BRAMs of its available hardware resources. Our future work will focus on the implementation of the complete Kyber scheme, its optimisation and resistance against side channel attacks.

ACKNOWLEDGEMENTS

This work is supported by Ministry of the Interior of the Czech Republic under grant VJ01010008.

REFERENCES

Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2017). Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2:4.

Basu, K., Soni, D., Nabeel, M., and Karri, R. (2019). Nist post-quantum cryptography-a hardware evaluation study. *IACR Cryptol. ePrint Arch.*, 2019:47.

Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. (2009). Keccak specifications. *Submission to nist (round 2)*, pages 320–337.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2014). (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36.

Chen, Z., Ma, Y., Chen, T., Lin, J., and Jing, J. (2020). Towards efficient kyber on fpgas: A processor for vector of polynomials. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 247–252. IEEE.

Chen, Z., Ma, Y., Chen, T., Lin, J., and Jing, J. (2021). High-performance area-efficient polynomial ring processor for crystals-kyber on fpgas. *Integration*, 78:25–35.

Dang, V. B., Farahmand, F., Andrzejczak, M., Mohajerani, K., Nguyen, D. T., and Gaj, K. (2020). Implementation and benchmarking of round 2 candidates in the nist post-quantum cryptography standardization process using hardware and software/hardware codesign approaches. *Cryptology ePrint Archive: Report 2020/795*.

Ferozpuri, A. and Gaj, K. (2018). High-speed fpga implementation of the nist round 1 rainbow signature scheme. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE.

Huang, Y., Huang, M., Lei, Z., and Wu, J. (2020). A pure hardware implementation of crystals-kyber pqc algorithm through resource reuse. *IEICE Electronics Express*, pages 17–20200234.

Langlois, A. and Stehlé, D. (2015). Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599.

Marotzke, A. (2020). A constant time full hardware implementation of streamlined ntru prime. In *International Conference on Smart Card Research and Advanced Applications*, pages 3–17. Springer.

Nejatollahi, H., Dutt, N., Ray, S., Regazzoni, F., Banerjee, I., and Cammarota, R. (2019). Post-quantum lattice-based cryptography implementations: A survey. *ACM Comput. Surv.*, 51(6):129:1–129:41.

NIST (2015). Fips pub 202 sha-3 standard: Permutation-based hash and extendable-output functions.

NIST (2016). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf.

NIST (2019). Computer security resource center (csrc): Post-quantum cryptography - round 3 submissions. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. Last accessed 04-March-2021.

Ricci, S., Malina, L., Jedlicka, P., Smekal, D., Hajny, J., Cibik, P., and Dobias, P. (2021). Implementing crystals-dilithium signature scheme on fpgas.

Roy, S. S. and Basso, A. (2020). High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Cryptol. ePrint Arch.*, 2020:434.

Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M., and Karri, R. (2020). Hardware architectures for post-quantum digital signature schemes.

Soni, D., Basu, K., Nabeel, M., and Karri, R. (2019). A hardware evaluation study of nist post-quantum cryptographic signature schemes. In *Second PQC Standardization Conference*. NIST.

Wang, W., Szefer, J., and Niederhagen, R. (2018). Fpga-based niederreiter cryptosystem using binary goppa codes. In *International Conference on Post-Quantum Cryptography*, pages 77–98. Springer.

Xing, Y. and Li, S. (2021). A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 328–356.