Adapting and Extending Selected Open-Source Hardware Implementations of Post-Quantum Cryptography Algorithms with Support for the CERG Hardware API

Alex Olihovik, *Student Member, IEEE*

***Abstract*— A VHDL implementation of the NewHope algorithm was modified to meet the application program interface (API) of the George Mason University (GMU) Cryptographic Engineering Research Group (CERG) team for post-quantum (PQ) algorithms. The API implementation increases resource utilization by a negligible amount, meets the same timing requirements as the original implementation, requires minimal increases in power utilization, and provides a fair and uniform interface for ease of testing and efficient use of hardware resources.**

***Index Terms*—API, Post-Quantum, Public Key, VHDL**

I. Introduction

Current cryptographic standards like the Rivest-Shamir-Adleman (RSA) cryptosystem rely on the inability of modern computers to factor large prime products efficiently. Some researchers estimate that a new class of devices capable of efficient computation of algorithms that can be simplified into a unitary matrix structure will be used to break current standards within the next few decades. These devices will utilize quantum effects of their underlying components to increase performance of operations such as prime factorization and will effectively nullify the resilience of current cryptosystems to quantum-based attacks. To meet the need of rising need for cryptographic standards that can defend against systems capable of quantum computing, post-quantum (PQ) algorithms that give little to no advantage to quantum-based attacks have been proposed.

Research into post-quantum algorithms is ongoing while many of the proposed algorithms compete to become the new standard. These algorithms differ in implementation, with no unifying application program interface (API) guiding the development of code bases presented to the cryptographic community. Such disparate coding structures have negative effects on hardware implementations as there is no clearly-defined interfaces for public, secret, and random data. This also makes use of a universal testbench for hardware applications more difficult to program as the testbench needs to be recoded for each new algorithm and gives no guarantee of efficient reuse of hardware, input and output data stream management, or fairness in comparison of algorithms.

Alex Olihovik is currently a student at George Mason University in Fairfax, VA and is employed by SAZE Technologies in Silver Spring, MD.

There is, however, an API by Ferozpuri et al. [1] that if followed would provide all the benefits of interconnectability, efficient hardware reuse, and fairness for hardware implementations among the various competing post-quantum algorithms. For this paper, the API put forth by the George Mason University (GMU) Cryptographic Engineering Research Group (CERG) was implemented using a post-quantum algorithm code base by Tobias Oder and Tim Güneysu in VHDL [2]. The algorithm targeted was a post-quantum key exchange mechanism (KEM) named the NewHope algorithm. This paper demonstrates and documents the development of adapting the reference NewHope algorithm implementation to meet the CERG API.

II. The GMU CERG Hardware API

The hardware interface proposed by the GMU CERG team standardizes the input and output specifications for the major operations in cryptography, such as encryption and decryption, signature generation and verification, and key encapsulation and decapsulation. The design is applicable to all post-quantum algorithms as it covers inputs and outputs for public, secret, and random data. The design enforces certain coding guidelines that provide for ease of interaction with other hardware cores implementing a similar interface, hardware reusability, and fairness in implementation among both different algorithms and the same algorithm.

Cores developed following the CERG API have interfaces and protocols like the AXI-Stream interface, such as in Figure 1. For each data line, there are additional signals that indicate when the core is ready to accept data, and when its output is valid. For instance, if an upstream producer of data flags its current output as valid, and the CERG-following API core flags its input port as ready, data is considered consumed by the core and the upstream producer can generate new data. Similarly, if the core outputs data and marks it as valid, and the downstream consumer marks its ready signal as high, the data is considered consumed and the core can generate new data for the consumer.

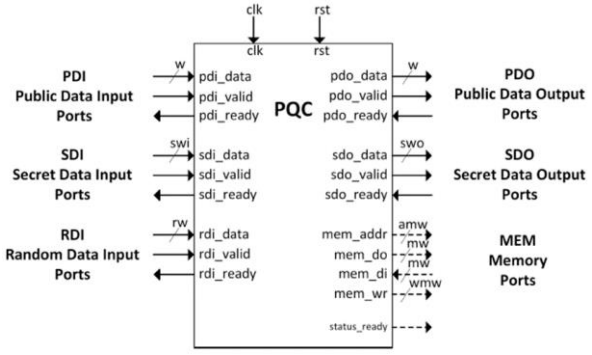

Figure 1. PQC Interface [1]

Common external circuits to such a core would include other AXI-Stream-enabled cores such as in Figure 2, or first-in-first-out (FIFO) registers to buffer data when data can't be consumed by a downstream core immediately.

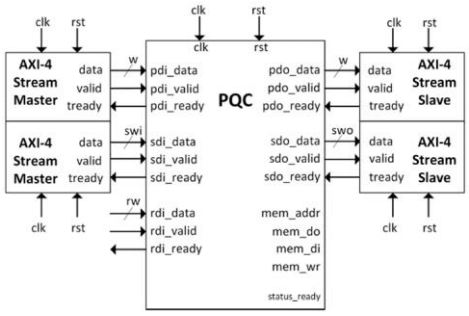

Figure 2. Core connected on input/output to other cores implementing the AXI-4 Streaming interface [1]

The API also specifies how to provide input and output headers, system parameters, and keys for each of the data ports, and indicates which fields are required for the different cryptographic functions, such as in Figure 3.

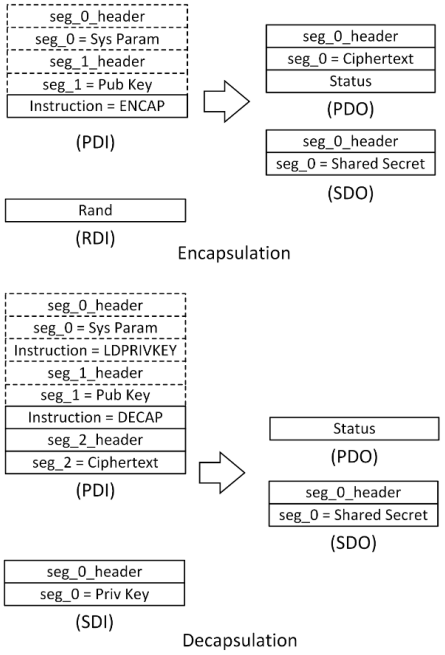

Figure 3. Format of data inputs and outputs for a key encapsulation and decapsulation [1]

III. The NewHope Algorithm

NewHope is a post-quantum lattice-based algorithm that was designed to be difficult for both modern and quantum computers to break. The NewHope algorithm was first developed from Peikert's Ring-Learning-With-Errors (RLWE) KEM which shares a similar structure [3]. The basic steps required to perform Peikert's algorithm first involves taking a system parameter **a** that would be fixed by previous compilation into software or hardware. Sample **s** and **e** from a distribution and use them in conjunction with a to generate **b** = **as** + **e**. The server then sends b to the client, which has sampled **s**', **e**', and **e**''. The client sends back **u**, which is closely resembles **b**, but requires a term **v**' to rectify any errors in **s**' and **e**' generation. If the sampled noise very large relative to the polynomial coefficients, both the client and server will generate different polynomials can't be resolved by sharing information of **b** and **u**. As the noise level decreases, the KEM becomes easier to break with simple linear algebra [4]. This gives a tradeoff between security and probability of failure.

Parameters: $q = 12289 < 2^{14},\ n = 1024$
Error distribution: ψ_{16}^n

Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \{0,\ldots,255\}^{32}$		
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$		
$\mathbf{s},\mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}',\mathbf{e}',\mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$		
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$	$\xrightarrow[1824\ \text{Bytes}]{m_a = \mathsf{encodeA}(seed,\hat{\mathbf{b}})}$	$(\hat{\mathbf{b}}, seed) \leftarrow \mathsf{decodeA}(m_a)$
		$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$
		$\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
		$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
		$\nu \xleftarrow{\$} \{0,\ldots,255\}^{32}$
		$\nu' \leftarrow \mathsf{SHA3\text{-}256}(\nu)$
		$\mathbf{k} \leftarrow \mathsf{NHSEncode}(\nu')$
		$\mathbf{c} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mathbf{k}$
$(\hat{\mathbf{u}},\bar{\mathbf{c}}) \leftarrow \mathsf{decodeB}(m_b)$	$\xleftarrow[2048\ \text{Bytes}]{m_b = \mathsf{encodeB}(\hat{\mathbf{u}},\bar{\mathbf{c}})}$	$\bar{\mathbf{c}} \leftarrow \mathsf{NHSCompress}(\mathbf{c})$
$\mathbf{c}' \leftarrow \mathsf{NHSDecompress}(\bar{\mathbf{c}})$		$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu')$
$\mathbf{k}' \leftarrow \mathbf{c}' - \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		
$\nu' \leftarrow \mathsf{NHSDecode}(\mathbf{k}')$		
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu')$		

Figure 4. The NewHope-Simple key exchange protocol [2]

The NewHope algorithm with protocol indicated in Figure 4 modifies Peikert's in ways that significantly increase KEM efficiency and decrease the overhead associated with generating highly accurate discrete Gaussian distributions as a source for noise samples [5]. To perform the NewHope algorithm, the system parameter **a** is created from a randomly-generated seed for each session. The seed is used as input to the SHAKE-128 function, which is a hash function that produces an arbitrarily long output, which is then parsed into a polynomial to generate system parameter **a**. The variables **s** and **e** are sampled from a centered binomial distribution Ψ_k of parameter k = 16. The client generates its polynomial in much the same way, with rectification given to the server by means of the variable **r**. The final key k is then hashed with the SHA3-256 function to create μ.

IV. Development on NewHope Reference Implementation

The reference implementation by Tobias Oder and Tim Güneysu was written in VHDL and provided source code for both the client and server implementing the NewHope algorithm separately. In order to implement the CERG API, the required inputs/outputs for a KEM were enumerated as public data input (PDI), secret data input (SDI), public data output (PDO), secret data output (SDO), and random data input (RDI) and mapped to the variables provided by the reference implementation. For the server, PDO was mapped to the seed and coefficients for the polynomial **b**, SDO was mapped to the key, PDI was mapped to the coefficients for the polynomials **u** and **c**, and SDI was mapped to the public key of the client. The client uses PDI, SDI, PDO and SDO, which maps to the seed and polynomial **b**, the private key of the client, the polynomials **u** and **c**, and the key, respectively. Other signals were added to satisfy the timing requirements for efficient data exchange and holding.

The pqc_wrapper allows for variable data input and output widths, which are set by integer generics. These generics control the sizes of the public, secret, random, and memory data inputs and outputs. Additionally, a generic std_logic named encap controls whether the core will perform key encapsulation or decapsulation. Following the API, once the core is instantiated in an architecture, it will only perform either encapsulation or decapsulation in a half-duplex mode of operation. The std_ logic encap controls a generate instruction in the architecture. If encap is asserted high, a NewHope_Server is created inside the pqc_wrapper and signals are routed accordingly. Similarly, if encap is asserted low, a NewHope_Client is created and routed.

The pqc_wrapper also meets the timing characteristics specified by the API. Pdo_data and pdo_valid signals are updated when the input pdo_ready signals are high, and similarly for pdi_data and associated signals.

The testbench for the PQC wrapper utilized code from the original NewHope VHDL implementation. The simulation runs for roughly 3 ms before producing keys from the client and about 3.5 ms before producing keys by the server. The testbench was also modified to output the keys to text files labelled “server_keys.txt” and “client_keys.txt”, which is useful for post-simulation comparison by a python script or other means to ensure the keys produced at the client and server match. In the example testbench provided, the keys for both the client and the server match each other before and after implementation of the pqc_wrapper. Additionally, the keys generated without the wrapper match the keys generated after using the wrapper. The number of keys written to the file is user-definable and is included only due to the length of the testbench run time.

The testbench was also modified to fit the data formats specified in the API. Specifically, for the server, the signals a_seed_server, poly_b_out, streaminga, DONE1, DONE2 and request_c are respectively concatenated and form pdo_data, while poly_u, poly_c, finalize and some padding data are respectively concatenated to form pdi_data. The generated key stored in the signal key_outa comprises the sdo_data output. One adaptation to this code that did not produce viable results was the attempt at removing the random seed that would normally be provided as input to the core by a secure random number generator [2], which is implemented as a signal stored in the NewHope_Server code on line 333 as a_seed_intern. The attempt involved providing this 256-bit input as the rdi_data input to the pqc_wrapper, but this failed in the implementation stage due to overutilization since the total design would have contained 320 I/O ports while the target device only contained 106 available for user I/O. From the client side, this wouldn’t have posed a problem as only 8 bits of the 256-bit random data is used at one time, but the Keccak 1600 core within the server requires the full 256-bit input to process. Further investigation into the Keccak 1600 core indicates that the random data and some padding is XORed with the current state and used as the input to a multiplexer but reducing this input to a number of I/O lines compatible with the target device was not performed to remove this incompatibility with the API. On other target devices, this might not be an issue, and as such there are comments in the code labelled “Note 1” where lines should be uncommented if this direct application of rdi_data is possible.

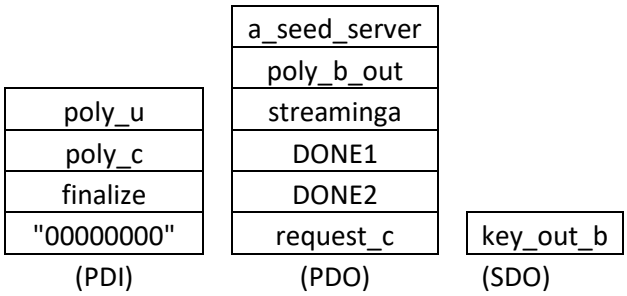

Figure 5. Format of Public Data Input (PDI), Public Data Output (PDO), and Secret Data Output (SDO) for the Server

For the client, pdo_data is constructed by consecutively concatenating poly_u_out, poly_c_out, streaming, request_b, and done_client, along with the signal w_pdo_extra that forms the padding output of pdo_data and removes warnings in the simulation when connected. Pdi_data is comprised of signals a_seed_client and poly_b with some padding by consecutive concatenation.Likewise, the generated key stored in the signal key_outa comprises the sdo_data output.

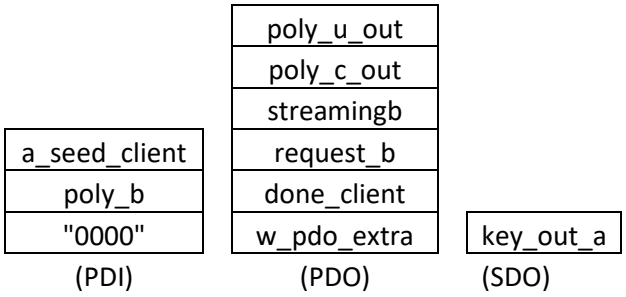

Figure 6. Format of Public Data Input (PDI), Public Data Output (PDO), and Secret Data Output (SDO) for the Client

Also included with the source code is a generically-sized random-access memory core, gen_ram.vhd, which is one example of an external circuit that can interface with the

pcq_wrapper. This code was adapted from an earlier implementation from another student implementing the BLISS algorithm.

V. Results

Post-synthesis results after using the wrapper show resource utilization increases by 19 LUTs for a 0.09% increase, 3 FFs for a 0.01% increase, 9 IOs for an 8.49% increase, and 3 BUFGs for a 9.37% increase. Post-implementation results after using the wrapper show resource utilization increases by 13 LUTs for a 0.07% increase, 4 FFs for a 0.01% increase, 9 IOs for an 8.49% increase, and 3 BUFGs for a 9.37% increase. Overall power utilization increases by 0.004 W as power allocated to clocks, signals, logic, and I/O changes by -0.001 W, 0.004 W, 0.003 W, and -0.002 W respectively. Timing reports indicate that the maximum clock frequency, 125 MHz, was the same before and after modification with new worst negative slack (WNS) of 0.01 ns which was decreased from 0.138 ns, worst hold slack (WHS) of 0.045 ns which was increased from 0.028 ns, and worst pulse width slack (WPWS) of 3.020 ns which remained the same.

VI. Conclusion

As expected for the implementation, there is little added resource utilization, power utilization, and no timing differences except for small latency increases when modifying the reference implementation to meet the CERG hardware API. The benefits are numerous as this increases compatibility with other cores implementing the AXI-4 steaming interface, offers portability into a universal testbench with standardized inputs and outputs, and provides fairness for benchmarking purposes. This implementation is one realization of the hardware API that could help evaluate many different algorithms for correctness, speed, and area efficiency in an upcoming post-quantum era for cryptography.

Acknowledgment

The author would like to thank Professor Kris Gaj for his insightful guidance throughout this project, excellent teaching abilities in the classroom, and willingness to accommodate his students' ever-shifting schedules and workloads.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.010 ns	Worst Hold Slack (WHS):	0.045 ns	Worst Pulse Width Slack (WPWS):	3.020 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	11338	Total Number of Endpoints:	11338	Total Number of Endpoints:	4749

All user specified timing constraints are met.

Figure 7. Setup, hold, and pulse width analysis

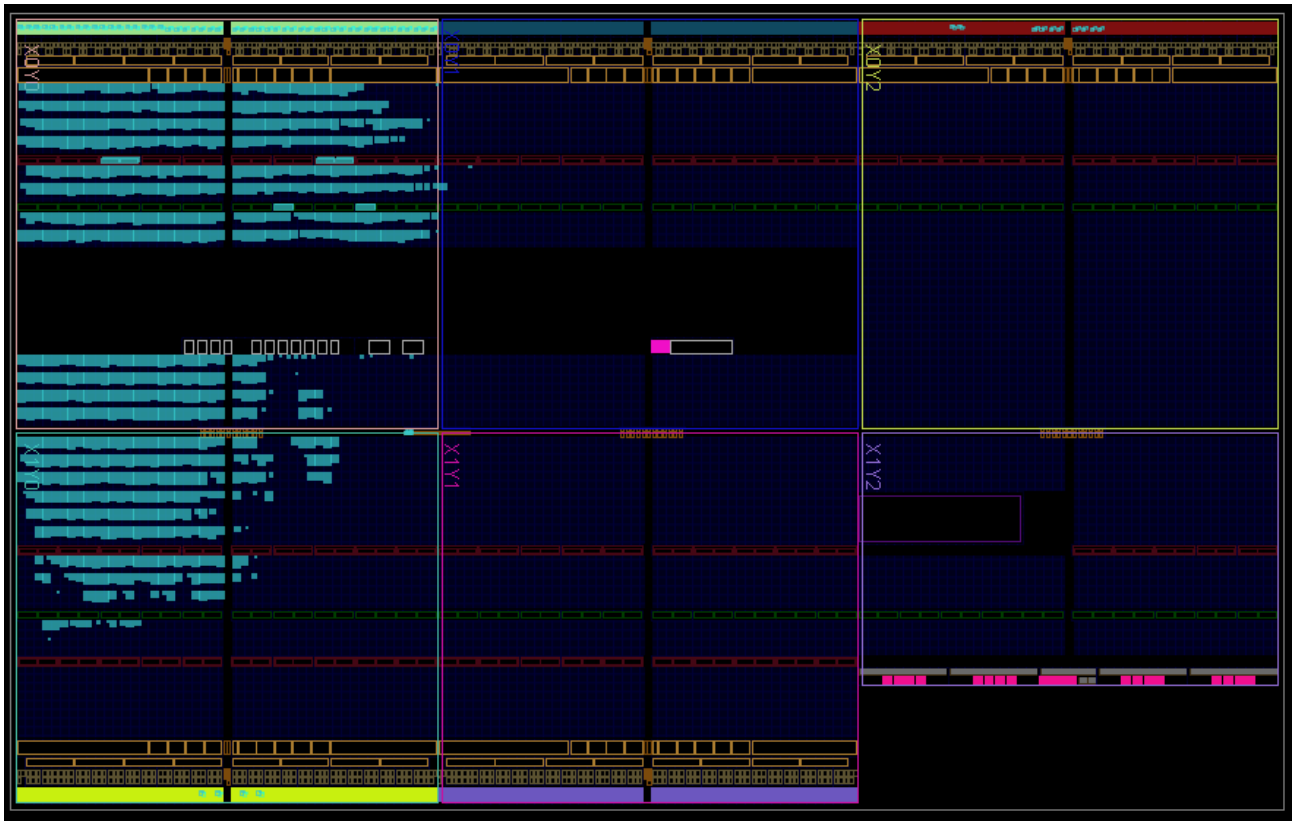

Figure 8. Implemented design

REFERENCES

[1] Ferozpuri, A., Farahmand, F, et al.: Hardware API for Post-Quantum Public Key Cryptosystems, https://cryptography.gmu.edu/athena/PQC/PQC_HW_API.pdf

[2] Oder, T., Güneysu, T.: Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs. Cryptology - LATINCRYPT 2017 - 6th International Conference on Cryptology and Information Security in Latin America, Cuba, September 20-22, 2017, Proceedings, 2017. https://www.ei.ruhr-uni-bochum.de/media/seceng/veroeffentlichungen/2018/04/16/newhope_fpga.pdf

[3] Alkim, E., Ducas, L., Poppelmann, T., Schwabe, P.: Post-quantum key exchange – a new hope. In: Proceedings of the 25th USENIX Security Symposium. USENIX Association (2016), document ID: 0462d84a3d34b12b75e8f5e4ca032869, https://cryptojedi.org/papers/newhope-20171212.pdf

[4] Alkim, E., Ducas, L., Poppelmann, T., Schwabe, P.: Post-quantum key exchange – a new hope. Cryptology ePrint Archive, Report 2015/1092 (2015), http://eprint.iacr.org/2015/1092

[5] Alkim, E., Ducas, L., Poppelmann, T., Schwabe, P.: NewHope without reconciliation (2016), https://cryptojedi.org/papers/newhopesimple-20171108.pdf