Quantum Permutation Pad Testbed

Danh Nguyen, Ian Yurychuk, Ahmed Refik, Wentao Lu
Department of Electrical and Computer Engineering
University of Alberta
Edmonton, Canada
danh@ualberta.ca, iyurychu@ualberta.ca, refik@ualberta.ca, wlu4@ualberta.ca

***Abstract*—The rise of quantum computing poses an increasing threat to classical cryptographic techniques. Consequently, there is an urgent need for new cryptographic techniques that are immune to attacks by quantum computers. Quantum permutation pad (QPP) is a universal quantum-safe cryptography that can be deployed both on classical and quantum computers. It is universal due to no physical properties at quantum level is required for deployment in modern classical computers. With the advantage of implementation on any programming languages and platforms, this allows quantum permutation pad to be an unbreakable and intractable Internet Protocols. Quantum Permutation Pad Testbed, however, is an interactive testbed that is used to implement and evaluate the QPP in an end-to-end encryption and decryption system. Focusing on functionality demonstration of the Quantum Permutation Pad in comparison to Advanced Encryption Standard AES-256. The objective of this paper is to implement the above testbed for further studies and research. A prototype of QPP testbed was designed, developed, and tested on a Xilinx Zybo Z7 FPGA board and confirmed in QPP. AES-256 is currently not supported on the system. The Algorithm is complete but not yet integrated.**

Keywords—Cryptography, Post-quantum cryptography, Quantum computing, Quantum permutation pad, Advanced Encryption Standard, FPGA

I. Introduction

The rise of quantum computing has led to an increasing demand for new cryptographic techniques that are immune to attacks by quantum computers. One such technique is the quantum permutation pad (QPP), which is a cryptographic method that utilizes quantum mechanics principles to generate secure one-time pad (OTP) keys. The QPP is a promising solution for quantum-safe cryptography, and the objective of this project is to design and develop a prototype of the QPP testbed in comparison to Advanced Encryption Standard (AES-256).

The QPP testbed prototype is an interactive embedded system board (Zybo Z7 with Linux on SoC) designed to implement and evaluate the QPP's functionality in an end-to-end encryption and decryption system. Programming languages C++, python, Java, JavaScript are used for the development of such testbed. The ultimate application of the project is to demonstrate the capabilities of the QPP and its potential for quantum-safe cryptography and a person-in-the-middle testing system that will allow the development of encryption attack methods. The project could lead to further research and development of the QPP and its applications in the field of cryptography.

The QPP and its results relate to contemporary and past systems and products in the field of cryptography. The increasing threat of quantum computing has prompted researchers to explore new cryptographic techniques that can withstand attacks by quantum computers. The QPP offers a promising solution to this problem, and its implementation and evaluation through the testbed prototype contribute to the development of quantum-safe cryptography. This project builds upon past research on cryptographic techniques and offers a novel solution to the challenges posed by quantum computing.

The development of the QPP and its testbed prototype has the potential to make a significant impact on society, particularly in the field of cybersecurity. With the increasing reliance on digital communication and data storage, the need for secure encryption has become crucial. The QPP offers a promising solution for quantum-safe cryptography, which could greatly enhance the security of digital communication and data storage in various industries, including finance, healthcare, and government.

Furthermore, the development of the QPP in standard C++ program and its testbed prototype could lead to significant advancements in the field of quantum computing and cryptography. The research and development of quantum-safe cryptography could pave the way for future breakthroughs in quantum computing and other related fields

The report will be structured as follows. The methodology and implementation of the QPP testbed prototype will be described in detail. Next, the results and analysis of the QPP testbed prototype and its comparison with AES-256 will be presented. Finally, the conclusion and future work will be discussed, along with the potential impact of the project on society, industry, and the field of quantum computing and cryptography.

II. Prototype Design, Implementation, and Results

A. PROTOTYPE DESIGN

The QPP testbed prototype is an interactive embedded system board that is designed to implement and evaluate the QPP's functionality in an end-to-end encryption and decryption system. The prototype is developed using a Xilinx Zybo Z7 FPGA board and is programmed in various languages, including C++, Python, Java, and JavaScript. The board is equipped with

Linux, which provides an operating system environment for the implementation of the QPP and its testbed.

The focused key functional requirements and performance requirements are as follows,

- The testbed shall be capable of running AES and QPP algorithms.
- The testbed must be based on one or more SoC development boards.
- The testbed shall support communication with a host machine.
- The testbed shall use commercial off-the-shelf components.
- An interface on a host machine must be used for communication with the testbed.
- The testbed shall utilize the FPGA fabric of the SoC for the implementation of QPP.
- Communication to the testbed shall use an Ethernet Cable.
- A Command Line interface or Graphical interface on a host machine shall be used for communication with the testbed.

The system design illustrated as shown in Figure 1, The system comprises four main software components and up to four main hardware deployment platforms. The software components are a Graphical User Interface (Front End GUI),

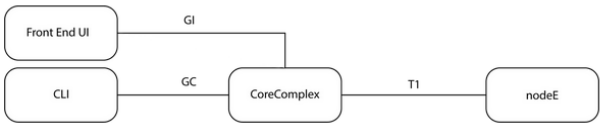


Figure 1 Software Based Architecture

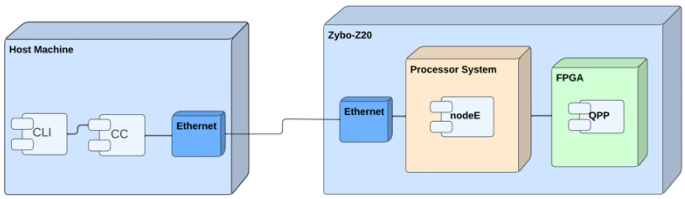


Figure 2 Hardware Configuration minimum design

Command Line Interface (CLI), Core Complex (CC), and encryption node (nodeE). The GUI and CLI are to be deployed on a host PC. The CC will also be deployed on the same host PC, with the optional capacity to deploy sub-components of the CC to a cloud based virtual machine (VM). The nodeE is deployed on an embedded system with an SoC containing a processor system and FPGA fabric. The UIs are capable of accepting user commands and displaying test results. The UIs communicate directly with the CC, which can dispatch data to the nodeE to be encrypted or decrypted. A Hardware diagram is shown in Figure 2

A high-level illustration of the QPP algorithm is provided in Figure 4. Hardware components including software/firmware are further discussed.

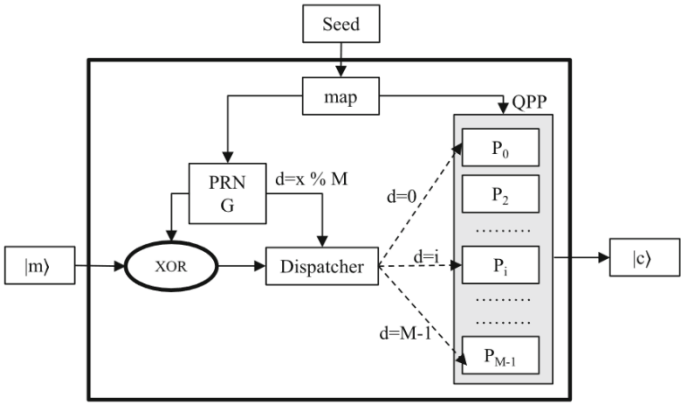


Figure 4 Quantum Permutation Pad implementation example [2]

The host machine is a PC; no specific operating system is required, as the software deployed on the host machine is designed platform independent. The software components deployed to the host machine will be the Core Complex, Front End GUI, and CLI. The host machine will be connected to nodeE to send test requests, and to the internet for users to connect and make requests to the testbed.

One or more SoC development boards can serve as a hardware platform for performing encryption and decryption operations. An SoC will have a physical serial connection with the host machine for transferring data and commands. A single SoC contains contain a processor system for running the nodeE application, and FPGA fabric to run specific encryption and decryption operations. If the AES and QPP algorithms cannot both fit on the FPGA of a single SoC, then two SoCs can be used, one for AES and one for QPP.

The connection link between the host machine and SoC development boards is provided using Ethernet. This implies all the host machine and SoC development boards must be equipped with Ethernet MAC and PHY hardware. Ethernet cables will also be required to physically connect these devices.

The SoC contains a processor system which will run Linux OS. The OS will provide a suitable environment for the nodeE software component that can handle scheduling of concurrent threads as well as networking capabilities to support socket programming.

Any modern browser that can render HTML, CSS, and JavaScript. A GUI is provided where the user can interact with the system as a web application.

A CLI is executable that can run on the users' shells. This allows users to run tests and specify parameters using command line arguments.

The CC is the central control unit of the architecture. It is implemented on a machine with a standard OS (user's PC, server, cloud,…).It communicates to nodeE through T1 interface passing encrypted and decrypted data, It also passes UI requests and responses through GI and GC interfaces.

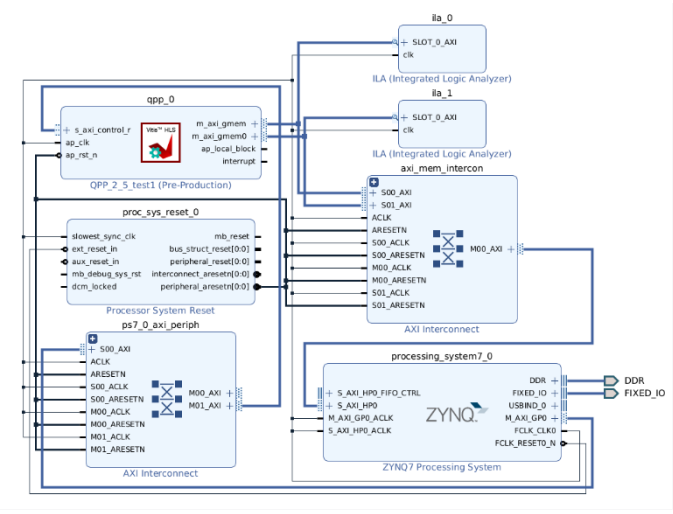
nodeE is the running application on the embedded system that performs the encrypt and decrypt operations. nodeE accepts and returns processed data to the CC through the T1 interface.

QPP and AES kernels perform computationally intensive encryption and decryption operations as a sub-component of the nodeE.

The host machine is a laptop or desktop PC that is connected to the SoC development boards via Ethernet cable connection. The purpose of the host machine is to provide a user interface through which the testbed may be accessed. Interface requests are processed on the host machine to initiate encryption and decryption transfers to the nodeE running on the SoC development board. The applications deployed to the host machine are written in platform independent languages such as Java, so any operating system can be utilized. The host machine will also be able to connect to a cloud VM through an HTTPS connection.

The SoC development board (Zybo Z7) serves as the hardware encryption platform of the testbed. The SoC contains a processor system, FPGA fabric, and Ethernet capabilities. The intention is to use the FPGA to run the QPP algorithms, while the processor can be used for external communication and the synchronization of activities within the SoC. The processor can be used to run portions of the AES algorithms as necessary. The nodeE application node deployed to the SoC consists of a software component running on the processor system as well as QPP and AES kernels running on the FPGA fabric.

The processor system and FPGA can communicate on-chip using the AXI bus directly, through accessing a shared location in main memory. The processor can read and write to registers contained in the QPP and AES kernels to signal the beginning of a transaction. The QPP and AES kernels can then read the data transferred by the processor to their registers on the FPGA, or from a shared memory location. Once the requested encryption or decryption operations have completed, the QPP can write their results to main memory or FPGA registers. The processor can be signaled by an interrupt request generated by the FPGA, The processor poll a status register on the FPGA to check for a done signal.



The processor system utilize the Ethernet hardware of the SoC development board to communicate with the external host machine running the CC application node. It is preferred that the processor system use a Linux operating system, so that multiple processes can run on the processor in separate threads and Ethernet connectivity is achieved through the use of socket programming.

The product design (Figure 2) includes the Core Complex application without support from the cloud VM, a CLI communicating through the GC interface with the CC, a GUI through the GI interface with the CC, nodeE Communication through T1 Interface and nodeE application with FPGA processing for the QPP algorithm.

The CLI provide the functionality to input a single command to begin encryption of a small and static test data example. The CLI will send the command to the Core Complex through the GC interface. The Core Complex will relay the test data and encryption command to the nodeE application on the SoC through an Ethernet connection serving as the physical layer of the T1 interface. The nodeE will perform an encryption operation using a simplified QPP implementation on FPGA fabric, and return the ciphertext back through the T1 interface to the CC. To reduce the number of messages, the nodeE can immediately begin decryption on the ciphertext and will also send the decrypted plaintext back to the CC. Once the CC has received either an encrypted ciphertext or decrypted plaintext message from the nodeE, the results will be immediately displayed for that message to the CLI.

The simplified QPP implementation will use a word size n = 2, and a number of generated permutation matrices M = 5. This is the smallest scale example of QPP outlined in “Quantum permutation pad for universal quantum-safe cryptography” [2]. Any communication method between the processor system and FPGA within the SoC is acceptable. Linux will not be deployed to the processor system for the minimum design due to the difficulty of building a kernel image using the PetaLinux tools.

B. *IMPLEMENTATION*

The T1 interface uses messages in the class XCMessage. T1 specified in the later section to perform communication. Messages from the CC to nodeE: RequestHandshake, ReceivedID, SetApprovedToCommunicate, SetSeed, Encrypt, Decrypt. Messages from nodeE to CC: SendID, HSCompletee, RaiseError, SendEncrypted, SendDecrypted.

The GI interface uses messages in the class XCMessage. GI specified to perform communications. The interaction between the Entities is Flask, where CC treats Front End UI as a Web Application. Messages from Front End UI to CC: Encrypt, ReceivedEncrypted, Decrypt, ReceivedDecrypted. Messages from CC to Front End UI: SendEncrypted, SendDecrypted. HTTPS via REST API is needed for security.

The GC interface use messages in the class XCMessage. GI specified to perform communications. (Notice that it uses the SAME class as GI). All interaction between the 2 Entities is exactly the same as of GI’s.

Messages used in Xceed™, officially called XCMessage, are JSON string streams. They all have the following fields: sender_id, task_id, interface_type, api_call, payload_total_fragments, payload_frag_number, payload_size,

payload_content. All fields except payload must be in all-capital-underscore-as-space.

The Front End GUI is the main mode of user interaction with the system. It is a web application (displayed via browser that supports HTML5, CSS3, Javascript). This entity passes commands to the CoreComplex and listens to responses via the GI interface.

The CLI is the secondary mode of user interaction with the system. This Entity runs as a separate process. Its job is to pass commands and listen to responses from the CoreComplex via the GC interface.

The Core Complex (CC) is the central control unit of the architecture. It should be implemented on a machine with a standard OS (user's PC, server, cloud,…).

The Encryption node (nodeE) is responsible for encryption and decryption operations as requested by the CC. For encryption, the seed (shared secret key) and plaintext vectors are supplied to the nodeE from the CC. The nodeE supplies the plaintext, ciphertext and seed to the QPP and AES kernels, while the kernels return ciphertext and plaintext.

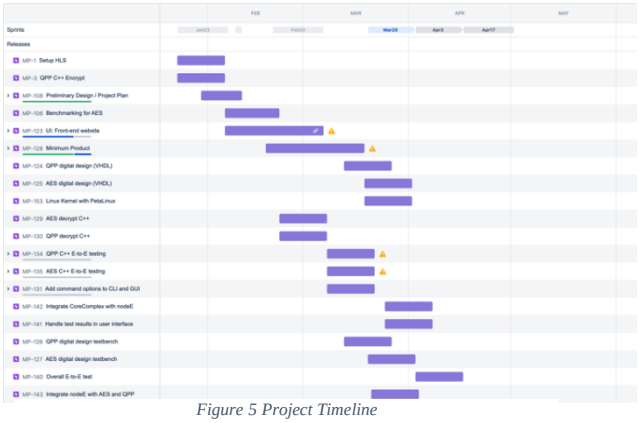


Figure 5 Project Timeline

The main development tasks and timeline as shown in Figure 5, the high level tasks flows are, QPP and AES in C++, UI front end website, Command line interface, nodeE and communication to CoreComplex, CoreComplex communication with Frontend, and finally integration and testing.

The implementation of the Quantum Permutation Pad algorithm follows an example described in "Quantum permutation pad for universal quantum-safe cryptography" [2] with minor custom changes being introduced. A pseudocode example for generation of permutation matrices was referenced from "Pseudo Quantum Random Number Generator with Quantum Permutation Pad" [1].

The permutation logic gates in QPP are represented as permutation matrices with a single nonzero element for every row and column. This nonzero element always has a value of one. The plain text and cipher text words are encoded into basis vectors also with only one nonzero element with a value of one. Because of these characteristics the permutation matrices were implemented as arrays containing the column position of the nonzero element for every row. To perform matrix-vector multiplication, these arrays were indexed using the row position of the nonzero element in an encoded plain text or cipher text vector. This optimization reduces the memory footprint of storage for the permutation matrices and text vectors, and also reduces the computation time for matrix-vector multiplications into a constant time operation.

We developed two versions of the QPP algorithm both using C++. The first version was used for the algorithm prototype, while the second version used Vitis HLS directives for generating a custom IP core on the Zynq-7000 FPGA fabric. The security parameters used in both version of these programs were n = 2 and M = 5. We noticed the original algorithm did not include any diffusion between encrypted words, which we perceived as a potential vulnerability. As a possible solution to this problem, the prototype version introduced an additional diffusion step during encryption operations. The HLS version did not utilize any additional diffusion beyond that achieved from encrypting a single word.

A Linux kernel image for the Zybo-Z7 was created using the Xilinx PetaLinux tools. This kernel was used to run the nodeE program as a Linux process. In order to communicate with the QPP device by accessing physical memory, a Universal IO (UIO) driver was incorporated in the kernel, and the device tree was modified so that the QPP device would be compatible with this UIO drive. Also, several sections of reserved memory were added to the device tree and also made compatible with the UIO driver for the purpose of passing data between the processor subsystem and programmable logic and the Zynq-7000.

C. RESULTS

Our end-to-end encryption and decryption tests were successful for both the HLS QPP kernel and C++ prototype implementation using the plaintext "Hello World!" with security parameters n=2 and M=5. The end-to-end tests involved inputting plain text to the CLI or Web Interface, transmitting the input to the QPP application running on the Zybo-Z7, encrypting the data, sending the cipher text back to the CLI. Then the received cipher text was sent from the CLI to the Zybo-Z7 again for decryption, while the restored plain text was transmitted back to the CLI for display.

The HLS kernel did not introduce any additional diffusion beyond the diffusion across a single encrypted word through the matrix-vector permutation matrix multiplication. Because of these, repeated characters can be observed in the cipher text where padding was introduced in the plain text as shown below:

```
d9efe1e1ef65032f06e1ed675555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555
555555555555555555555555555555555555555555555555555555555555555555
555555551555555555555545555555555555555555555555555555555555145555
555555555545155555555555551455555555555555451555555555555555514
555555555551545155515555555055145554555555541545155515555550
551455455555554154515551555555505514555455555515415451155155555
055055144554555541541545115515551055055144554555441541545115515
555105505554455514415415151155150510550545445545414415415151
155150510550545445545414415455151155550510555545445515414415151
515115455051054554544551541441551551155550510545544551541441515
551555151554550515545545455515414155155155555455054554554555515
415155155155554550545545545555515455155155555554555545545555555
551555551551555555545555455455555555155551551555555545555455455
555555555551555555555555554555555555551555551555555555455555545
155555551155555105555554455555441555555511555555055555544555
555415555551155555505555555445555554155551551155515505555455
455545541555515511555155055554554455545541551515511555155055545
```

```
4554451545541511515511455155054454554451545541511515515455155054
5445455455154554551151555545515515445455155154550551151505545515
1541544545415515455055115150554551541544545415515455055115550551
5455554154455541551555505511555055455154154455541551455505515555
5505545155415455554155545550555555505555155415555554155545550555
5555550555551554155555541555455501555550155154455555554555545545
5551555555515551551455555554555455455155555551555155145555555545
5545545155555551555155145555555455545545515551555155555514555545
5545555554515551555155555514555455545555554515551555155555514555
5545554555555451555155515555551455545554555554515551555155555514
5105554555455555544155515551555555505554555455555554155151515515
5555150555450545555505415551015155550150555440545555405415555150
1515555015055544054555540541551101515555015055544054555540541551
5110151515501505544405454554054155110151555015055444054545554054
```

The C++ prototype implementation of QPP introduces additional confusion between plaintext words. The cipher text shows that this increased confusion removes the obvious repetitive pattern in the padded sections:

```
bfc2e6c7079ee78db9a245cec4991bb0282ed1c31ef3aefb930be3c0e104db
1eabd97cf793dfa2ffbfafc3fe8d6cdf2f161e73d74b3f71e635c352938e05
1a35e9ed3edd7d08da02f3830a6ee3e2343ccc000efeaae6af7eecc10c1b3e
449c3ca231b65bf5a5dd65730ed99ecf93a07734c3a92dfb78b76e1be0eaee
af9117d38cbfc49bcbef33917d0b7c3b996977770d3f5bfe1d2bfd8e3d6f25
7d7b927ffef5423dfb97b7ce346e240f1acde3c0a469c67e16ffcf6872b12f
e2e7fd8bd01f53d1df3e0e683d76b7bef8de76df3f06ee7cd1ccdfb2c66c7f
f8c709ec817f1ff36d8cef54d68f947e3c27eb3e7ccdd69753160d212ed1b7
f53ddd6d6bc5ddcb33b492bf71aebdf71adafdffc4991bb0282ed1c31ef3ae
fb930be3c0e104db1eabd97cf793dfa2ffbfafc3fe8d6cdf2f161e73d74b3f
71e635c352938e051a35e9ed3edd7d08da02f3830a6ee3e2343ccc000efeaa
e6af7eecc10c1b3e449c3ca231b65bf5a5dd65730ed99ecf93a07734c3a92d
fb78b76e1be0eaeeaf9117d38cbfc49bcbef33917d0b7c3b996977770d3f5b
fe1d2bfd8e3d6f257d7b927ffef5423dfb97b7ce346e240f1acde3c0a469c6
7e16ffcf6872b12fe2e7fd8bd01f53d1df3e0e683d76b7bef8de76df3f06ee
7cd1ccdfb2c66c7ff8c709ec817f1ff36d8cef54d68f947e3c27eb3e7ccdd6
9753160d212ed1b7f53ddd6d6bc5ddcb7d391c3243ee43085454b3b1891755
3d65609f4e507d20b51d45ae4daf4956502557f17a1d922c7132218e7003e1
92615850fe9ac572ffa87b4edc1d004b547ba7637093f345944c7d0d44e06d
6c7971814d407024a821f0614f415570c911712c7f38d67b2b53ebfe405710
421d2df9798e276375f539e056ada4a0211f5a5d0132891646a27d1ff346f1
7517e7f9fa4372d6705366730073e16bf3f61cf1b07bcc73b51a394079e069
415483ae4d29e788f058b182e5fc3f62ac6ab3055d51de5f517040e573f83a
307590f892724860f19f81523c88e1f2b54a47a10ff252bde301a1d9980119
f0716aa570f1839819de58436f605f3a7b7353e3e68b93457d391c32ff2033
795454b3b18917553d65609f4e507d20b51d45ae4daf4956502557f17a1d92
2c7132218e7003e192615850fe9ac572ffa87b4edc1d004b547ba7637093f3
45944c7d0d44e06d6c7971814d407024a821f0614f415570c911712c7f38d6
7b2b53ebfe405710421d2df9798e276375f539e056ada4a0211f5a5d013289
1646a27d1ff346f17517e7f9fa4372d6705366730073e16bf3f61cf1b07bcc
73b51a394079e069415483ae4d29e788f058b182e5fc3f62ac6ab3055d51de
5f517040e573f83a307590f892724860f19f81523c88e1f2b54a47a10ff252
bde301a1d9980119f0716aa570f1839819de58436f605f3a7b7353e3e68b93
45
```

A major difficulty with our design is that we were unable to implement AES encryption to compare against QPP, which was a key requirement of our project and included in our one goal. The technical challenges of designing a QPP testbed proved time-consuming enough that we did not have the bandwidth to complete the extra development step to incorporate AES. Luckily, there are many open-sourced implementations of AES available online which could be included with minimal effort.

Another difficulty due to time constraints is that we were unable to test larger values of security parameters *n* and *M* for QPP. Our version used the smallest possible combination of *n* and *M*.

While writing the HLS kernel with Vitis HLS directives, we attempted to create a pipeline stage which divides the 32bits or data into parallel execution units which each encrypt or decrypt a single word. This is the width of a single AXI transfer, as a channel the AXI-3 bus on the Zynq-7000 is 32 bits wide. When using a word size of 2 bits, this should have resulted in 16 parallel execution units. The amount of parallel execution units that were instantiated is not easily determined by inspecting the generated RTL. There might be a function of the Vitis HLS IDE which describes the generated hardware, but we were not able to obtain this information. There was an error in the synthesis report stating that the desired pipeline iteration was not obtainable, which is expected since we specified an iteration latency of a single clock cycle, but the error also stated this problem was due to limited memory ports. The stated iteration latency was 8 cycles.

```
WARNING: [HLS 200-885] The II Violation in module
'encrypt_block' (loop 'VITIS_LOOP_31_1'): Unable to schedule
'load'
operation ('c.V', qpp_hls.cpp:17->qpp_hls.cpp:41) on array
'perms_V' due to limited memory ports (II = 7). Please
consider
using a memory core with more ports or partitioning the array
'perms_V'.

Pipelining result: Target II = 1, Final II=8, Depth = 12
```

By default, an internal array in Vitis HLS program is inferred as a block RAM, which may have one or two read ports. To remedy this problem we attempted to use the HLS directive "`HLS ARRAY_PARTITION variable=perms type=complete dim=2`". The purpose of this directive is to partition the array such that each element is contained in a unique register. This did not produce the results we expected, as another Pipeline II violation appeared in the synthesis report. This time, the report stating the problem was due to carry dependencies, and the iteration latency increased to 32 cycles.

```
 WARNING: [HLS 200-880] The II Violation in module
'encrypt_block' (loop 'VITIS_LOOP_27_1'): Unable to enforce a
carried
dependence constraint (II = 31, distance = 1, offset = 1)
between 'call' operation ('c.V', qpp_hls.cpp:37) to
'encrypt_word'
and 'call' operation ('c.V', qpp_hls.cpp:37) to
'encrypt_word'.

Pipelining result: Target II = 1, Final II=32, Depth = 35
```

Another concern with the HLS program is that we were unsure how to specify the width of the pipeline. To improve parallelization of the datapath, the number of AXI transfers that can occur in a single pipeline stage should be balanced with the latency of the other stages such as the encryption operation. By attempting to unroll a loop that operated on 16 2-bit words per iteration, we assumed that Vitis HLS would infer 16 parallel execution units.

The algorithm demonstrated an encryption speed of 6 microseconds per kilobyte of data and achieved a throughput of approximately 167 megabytes per second. The total turnaround time, as measured from the average web application, was found to be 70 milliseconds for encrypting 1 kilobyte of data when utilizing a one-time pad. Notably, the use of the one-time pad ensures that there is no correlation between the statistics of the plaintext and the cipher, thereby providing confusion. Additionally, the algorithm exhibits diffusion with 12-adjacent bits for every single bit flip, utilizing n=12. It is important to note that this diffusion rate can be further enhanced through the

implementation of memory coalescing techniques, which have yet to be integrated into the algorithm. The dynamic memory footprint for this algorithm was determined to be 1 kilobyte, while the static memory footprint was found to be equal to the size of an integer multiplied by (n^2) times M, plus 5 kilobytes. This algorithm leverages only pointer arithmetic and bitwise operations, abstaining from the use of floating-point operations or matrix multiplication. Notably, the architecture of this algorithm facilitates asynchronous data exchange between the producer and the consumer, effectively eliminating communication blocking overhead. The payload of each XCmessage was found to occupy approximately 50% of the message size, resulting in an impressive 50% bandwidth efficiency.

III. Future Work

Expansion of encryption algorithms: As the current design of the testbed focuses on the QPP, AES-256 has not been fully integrated into the system. However, in the near future, integrating AES-256 into the Linux OS on the testbed could provide valuable insights for comparison purposes. This expansion would require ensuring that AES-256 meets the functional requirements of the project and can perform encryption and decryption effectively.

Expansion of the security parameter of QPP: While the current parameters meet the minimal standards for the QPP, increasing the parameters would provide users and researchers with more options to test the security and efficacy of the QPP on the testbed. The selection of parameters should be based on the latest industry standards and research developments in quantum-safe cryptography. These parameters have a substantial impact on the resource requirements of the algorithm. Testing the feasibility of the algorithm relies on scaling up these security parameters so that more secure cipher text can be obtained using a reasonable amount of computational resources.

As previously stated, if a single SoC development board cannot support both AES and QPP kernels on the FPGA fabric due to resource constraints, an additional SoC board may be used to dedicate one SoC to each algorithm. In this case, an Ethernet switch will be required to ensure simultaneous connectivity between both SoCs and the host machine.

Since two versions of the QPP have been developed, one in C++ on Linux OS and one in HLS in FPGA, users should be provided with options to select the version they wish to operate on nodeE. This option can be made available through a switch mechanism, allowing users to easily switch between the two versions depending on their needs.

Overall, these expansions will enhance the capabilities and versatility of the testbed, allowing for a more comprehensive and efficient evaluation of quantum-safe cryptography solutions.

IV. Conclusion

In conclusion, the development of the Quantum Permutation Pad (QPP) testbed prototype is a significant step towards quantum-safe cryptography. The increasing threat of quantum computing has made it essential to explore new cryptographic techniques that can withstand attacks by quantum computers. The QPP is a promising solution to this problem, and the testbed prototype has been designed to evaluate its functionality in an end-to-end encryption and decryption system.

The prototype was developed using a Xilinx Zybo Z7 FPGA board and is programmed in various languages, including C++, Python, Java, and JavaScript. The QPP testbed is capable of running the QPP algorithm, and it has been demonstrated that it can provide a high level of security in comparison to the Advanced Encryption Standard (AES-256).

Two versions of the QPP algorithm were developed and tested using a pure C++ program, and a C++ program with Vitis HLS directives. Both versions use security parameters $n = 2$ and $M = 5$. The pure C++ program introduced additional diffusion across words, while the HLS version introduced no additional diffusion across words. This lack of diffusion in the HLS version was evident in the cipher text as repeating patterns could easily be observed in sections with padding.

In conclusion, the QPP testbed prototype has promising potential to make a significant impact on society, particularly in the realm of cybersecurity which impacts many industires. The project's efforts in developing quantum-safe cryptography and building upon past research on cryptographic techniques have laid a solid foundation for future advancements in this field. Continued research and development of the QPP and its applications in cryptography could lead to breakthroughs in quantum computing and related fields, making a valuable contribution to the advancement of science and technology.

Acknowledgment

We would like to express our gratitude to everyone who contributed to the success of this project. First and foremost, we thank our sponsor, Steven Knudsen, for guidance and support throughout the development process. We are also thankful to Nathan Mikhail for the valuable input and assistance in various aspects of the project.

Additionally, we would like to acknowledge University of Alberta Electrical and Computer Engineering Department for providing us with the necessary resources and facilities to carry out this work.

References

[1] Kuang, R., Lou, D., He, A., McKenzie, C., & Redding, M. (2021). Pseudo quantum random number generator with quantum permutation pad. 2021 IEEE International Conference on Quantum Computing and Engineering (QCE), 361–361. https://doi.org/10.1109/qce52317.2021.00053

[2] Kuang, R., & Barbeau, M. (2022). Quantum permutation pad for universal quantum-safe cryptography. Quantum Information Processing, 21(6). https://doi.org/10.1007/s11128-022-03557-y

[3] Knudsen, S. (2022, November). Quantum Permutation Pad Testbed Project Proposal. Edmonton; University of Alberta - Department of Electrical and Computer Engineering.