Quantum Encryption and Decryption in IBMQ Systems using Quantum Permutation Pad

Maria Perepechaenko and Randy Kuang
Quantropi Inc., Ottawa, Canada
Email: maria.perepechaenko@quantropi.com; randy.kuang@quantropi.com

Abstract—We present a functioning implementation of Kuang *et al.*'s Quantum Permutation Pad (QPP) using the Qiskit developmental kit on the currently available International Business Machines (IBM) quantum computers. For this implementation, we use a pad with 28 2-qubit permutation gates that provides 128 bits of entropy. In this implementation, we divide the plaintext into blocks of 2-bits each. Each such block is encrypted one at a time. For any given block of plaintext, a quantum circuit is created with qubits initialized according to the given plaintext 2-bit block. The plaintext qubits are then acted on with 2-qubit permutation operators chosen from a 28-permutation QPP pad. Due to the inability to send qubits directly, the ciphertext qubits are measured and transmitted to the decrypting side over a classical channel. The decryption can be performed on either a classical or quantum computer. The decryption uses an inverse Quantum Permutation Pad with the Hermitian conjugates of the corresponding permutation gates used for the encryption. We are currently working on advancing the implementation of QPP to include additional steps for security and efficiency.

Index Terms—quantum communication, quantum encryption, quantum decryption, quantum security, secure communication, QPP, Qiskit, International Business Machines Quantum (IBMQ)

I. Introduction

With the recent development in the field of quantum computing, increased interest in the field of quantum encryption and quantum key distribution came to light. An exciting new question is whether it is possible to establish direct secure communication between two quantum computers emerged. Indeed, at the 2022 Inside Quantum Technology event a so-called "killer app" has been discussed that would have the power to connect quantum computers, create quantum networks, and perhaps even quantum internet. That would require the secure key establishment and encrypted communication between various quantum computers that is efficient, available, and flexible. Current quantum alternatives to the asymmetric cryptographic algorithms, which are used mainly to securely distribute keys for future symmetric encryption, are Post-Quantum Cryptographic Algorithms and Quantum Key Distribution [1]-[5] The widely used symmetric encryption scheme, AES, was shown to be secure against quantum attacks, provided doubled key sizes [6]-[9]. There have been several quantum implementations of AES proposed in the last few years [10]-[14]. Grassl *et al.* presented a quantum implementation of the AES which was later improved by Kim *et al.* [10], [11]. More recently Langenberg *et al.* further reduced the number of quantum gates required for the for SubBytes operation [12]. In 2021, Wang *et al.* presented an improved implementation of the AES-128 in quantum system that reduces the cost of qubits by 224 compared to Langenberg *et al.*'s work [13]. They showed that the cost of implementing AES-128 on a quantum system is 18040 Toffoli gates, 101174 CNOT gates, and 1976 X gates on 656 qubits. In 2020 Kuang and Bettenburg proposed a unified symmetric encryption scheme, called the Quantum Permutation Pad (QPP) [15]. That is, a symmetric encryption scheme that can be implemented on both quantum and classical devices allowing for secure communication between two quantum computers, two classical computers, and a classical and a quantum computer.

In this paper, we will show a functional implementation of the QPP that uses a significantly smaller number of quantum gates, compared to AES, while providing 128-bits of entropy. The said implementation is done on the currently available publicly accessible quantum computers. The main difference between the quantum implementation of AES and QPP is that QPP does single-round encryption that depends on the secret key. QPP also differs from the quantum one-time pad schemes such as those presented by Ambainis and Smith [16] and Boykin and Roychowdhury [17]. In QPP the classical binary key is used to create the permutation pad of quantum operators that are used for encryption. The key size is the same for any massage and is only chosen according to the desired size of the permutation pad. Such a quantum pad is in the sense a key itself since it carries private secret information that should not be revealed but the nature of such a key is fully quantum. On the other hand, for perfect encryption with the quantum one-time pad, a key of the size $2n$ bits is required to encrypt n qubits. The construction consists of applying two classical one-time pads in two different bases, the standard basis, and the diagonal basis. For QPP the basis is not changed, in fact, the Hadamard gates are not used in the implementation.

Manuscript received July 5, 2022; revised November 17, 2022.
Corresponding author email: maria.perepechaenko@quantropi.com
doi:10.12720/jcm.17.12.972-978

II. Quantum Permutation Pad Implementation

Quantum Permutation Pad or QPP is a is a symmetric cryptographic algorithm that leverages a set of chosen quantum permutation gates or $2^n \times 2^n$ matrices expressed in n-qubit computational basis [15], [18], [19]. Since quantum operators are unitary and reversable, QPP can be used to encrypt plaintext or plain qubits to produce ciphertext or cipher qubits and its conjugate transpose QPP† can be used to decrypt the ciphertext or cipher qubits into plaintext or plain qubits. In this work we refer to QPP† as the Inverse Permutation Pad. This makes QPP a universal symmetric cryptographic algorithm that applies to both classical and quantum computers. Classical implementations with 8-bit permutation matrices have been reported in [20]-[22]. The first quantum implementation of a toy example in IBMQ quantum computers has been reported in [19]. In this paper, we are extending the work in [19] to achieve 128-bits of entropy.

As we have discussed in [19], the permutations used for QPP are elements of the symmetric group S_n. That is, a group of permutations of n elements. The order of S_n is $n!$, making each permutation provide $\log_2(n!)$ bits of Shannon information entropy.

Our implementation is done using 2-qubit permutations, in other words, each 2 qubits are acted on with a single permutation operator as we depicted in [19]. Since 2 qubits generate 4 possible states, each 2-qubit permutation is an element of the group S_4. There are at most $4! = 24$ such 2-qubit permutations. Each such permutation supplies $\log_2 24 \approx 4.58$ bits on entropy. Thus, to achieve 128 bits of entropy, the Permutation Pad must consist of 28 permutation gates. Note that there are at most 24 unique permutation gates, thus, we allow for repetitions in the pad.

Here we summarize the implementation as follows:

1. We suppose that communicating parties have pre-shared a secret key.
2. The encrypting party generates 24 distinct permutation gates by specifying corresponding permutation matrices row-by-row and then converting them to quantum operators.
3. To generate a Permutation Pad with 28 permutation operators, a random list of length 28 with indices of permutation operators is created.
4. The plaintext is randomized and broken into 2-bit blocks.
5. For each 2-bit block of randomized plaintext we use Qiskit commands to create the encryption QPP circuit and encrypt each such block, producing ciphertext qubits. The ciphertext qubits are then measured.
6. The classical binary ciphertext is broken into blocks of 2-bits each. For each such block we create a decryption QPP† circuit for decryption.

Detailed implementation is discussed in the next section Results and Discussion.

III. Results

In this paper, we present an implementation of Kuang *et al.'s* Quantum Permutation Pad (QPP) with 2-qubit permutation operators on the IBM quantum systems 'ibmq_manila' and 'ibmq_bogota' using the Qiskit software development kit. We will demonstrate encryption of the image of Schrödinger's cat, as shown in Fig. 1. The decryption can be done using a quantum computer or a classical computer.

Fig. 1. An image of a cat that will be encrypted by the encrypting party using QPP and decrypted by the decrypting party using the Inverse Permutation Pad. We name this image in the computer directory as "cat.png".

A. QPP Generation

In [19] we presented a way of generating permutation operators as a composition of basis quantum gates. In this paper, we present another way of generating the permutation operators by defining matrices corresponding to the respective permutations and converting them to Qiskit operators. For instance, one might specify a matrix that maps the state $|0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ to $|2\rangle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $|1\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ to $|3\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$, $|2\rangle$ to $|1\rangle$, and $|3\rangle$ to $|0\rangle$. Such permutation matrix is

$$M = \begin{bmatrix}0 & 0 & 0 & 1\\0 & 0 & 1 & 0\\1 & 0 & 0 & 0\\0 & 1 & 0 & 0\end{bmatrix}.$$

To convert matrix M to a quantum operator, a simple Qiskit command `Operator([[0,0,0,1], [0,0,1,0], [1,0,0,0], [0,1,0,0])` can be used.

To generate all possible permutation operators, the communicating party *A* first starts with an empty list and then adds all such permutation matrices *M* converted to operators to said list. Let such list be named `Permutations = []`. The matrices can be specified one by one by inputting them row by row. For 2-qubits, there are total 24 permutation gates. Suppose that *A* defines each matrix manually row by row then *A* will simply write the command

```
Permutation_matrix=np.array([[a0,a1,a2,a3],[b0,b1,b2,b3],[c0,c1,c2,c3],[d0,d1,d2,d3]])
Permutations.append(Operator(Permutation_matrix)))
```

for every permutation matrix where `a_i, b_i, c_i, d_i` are either 0 or 1 depending on the permutation matrix. We assume that *A* defines permutation gates one by one in some universal way that is known to everyone. For instance, starting with permutation (0), (0,1), (0,2), …, (0,3,2,1) written in cyclic notation. This will ensure that

the two communicating parties will be able to generate respective inverses of their Permutation Pads.

To start the encryption process, *A* still needs to generate a Permutation Pad with 28 permutations for 128-bits of security. One way of doing so is to generate a list of length 28 consisting of positions of the permutation operators in the list `Permutations`. Then during the encryption procedure, the permutation gates will be called from the Permutation Pad based on their corresponding position in the list `Permutations`. A simple code to generate a random list using python is as follows

```
Permutation_Pad= []
seed(5)
for _ in range(28):
        value = randint(0, 24)
   Permutation_Pad.append(value)
```

Note that seed is used to generate random values for the Permutation Pad. Such seed is a pre-shared secret to ensure that *B* precisely creates the Inverse Permutation Pad for decryption.

For a general case of n-qubit permutation gates, specifying permutations row by row will not be practical. Instead, clever algorithms such as Fisher-Yates can be used to produce all 28 permutation gates and place them into the pad array.

B. Encryption Procedure

Once the Permutation Pad is defined, the encryption procedure begins by converting the PNG file shown in Fig. 1 into a bitstring. Such bitstring is the plaintext to be encrypted. To account for the limitations of the existing available quantum computers, the plaintext is broken into blocks of two bits each. The plaintext will be converted to qubits and encrypted block by block with the permutation operators taken from the Permutation Pad one by one. That is, the first 2-bit block is converted to 2 qubits and encrypted with the first permutation from the Permutation Pad, the second with the second, and repeating in this pattern. Once *A* reaches the end of the Permutation Pad they continue again from the first permutation in the pad.

More precisely, after the plaintext is broken into chunks of 2 bits each, the encryption procedure is performed for each chunk separately using "for" loop. Given a single chunk of the plaintext, party *A* creates a quantum circuit for the corresponding chuck that consists of two qubits and two classical bits. Party *A* initializes the qubits according to the ciphertext block bits as described in detail in [19]. We denote such qubits that are initialized according to the corresponding plaintext block as plaintext qubits. *A* then applies the permutation operator chosen from the Permutation Pad to the plaintext qubits. Once the permutation operator is applied to a block of plaintext qubits, it successfully converts them to a block of ciphertext qubits. The final step in the encryption procedure is to measure the qubits and store the highest probability result in the binary file. Note that each circuit needs to be transpiled at the end to be successfully run on a quantum device. We include the source code for the encryption part in Fig. 2, and a sample transpired circuits for the message block at the position $x = 96$ in Fig. 3. The reader can find the sample measurement result of the ciphertext block at the position $x = 96$ in Fig. 4.

```
#code to convert PNG "cat.png" into a bitstring
out = BytesIO()
with Image.open("cat.png") as img:
    img.save(out, format="png")
image_in_bytes = out.getvalue()
message_cat = "".join([format(n, '08b') for n in image_in_bytes])

# message is bitstring of cat.png
message = message_cat
chunk_size = 2          #the size of each block is 2 bits/qubits
#breaking the message into blocks of 2
message_chunks = [message[i:i+chunk_size] for i in range(0,
len(message), chunk_size)]

list_of_ciphers = []

# plaintext is encrypted block by block
for x in range(len(message_chunks)):
#the qubits are initialized to the plaintetx bits
    state_vector = message_chunks[x]
    qc = QuantumCircuit(2, 2)
    qc.initialize(Statevector.from_label(state_vector))
    qc.barrier()
#permutation is applied to every chunk from the pad and the result is
measured
    j = Permutation_Pad[x%len(Permutation_Pad)]
    qc.append(Permutations[j], range(2))
    qc.barrier()
    qc.measure([0,1], [0,1])
#the circuit is transpiled
    qc = transpile(qc,
basis_gates=["u3","u2","u1","cx","id","u0","u","p","x","y","z","h","s"
,"sdg","t","tdg","rx","ry","rz","sx","sxdg","cz","cy","swap","ch","ccx
","cswap","crx","cry","crz","cu1","cp","cu3","csx","cu","rxx","rzz","r
ccx","rc3x","c3x","c3sqrtx","c4x"], optimization_level = 3)


#execute the circuit and store the final state in the list of
ciphertext chunks
    provider = IBMQ.load_account()
    qcomp = provider.get_backend('ibmq_bogota')
    job = execute(qc, backend=qcomp, shots = 1024)
    job_monitor(job)
    result = job.result()
    counts = result.get_counts()
    list_of_ciphers.append(counts.most_frequent())   #will only append
results with highest probability


# the ciphertext is stored in a binary file
ciphertext_one_str = "".join(list_of_ciphers)


file_cipher = open("ciphertext_to_send.bin","wb")
data = bytearray(ciphertext_one_str.encode("ascii"))
file_cipher.write(data)
file_cipher.close()
```

Fig. 2. The source code for the encryption procedure.

Fig. 3. Transpiled encryption circuit of the message block at position x = 96.

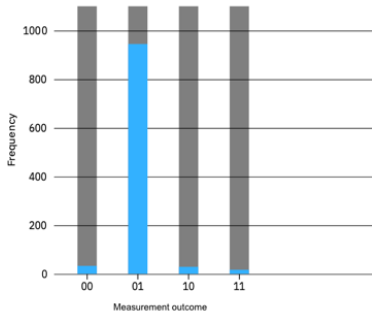

Fig. 4. Measurement result of the ciphertext block at the position x = 96. That is, encrypted plaintext block at the position 96.

C. Transmission of the Ciphertext

Ideally, given a free publicly accessible quantum channel, *A* would keep ciphertext qubits in their original form without measuring them, and transfer them to *B* over the said quantum channel. However, due to the lack of such channels today, *A* sends the measured binary

ciphertext file to *B* through any classical channel, even as an attachment in an email.

Suppose that the adversary C was able to obtain communication records between *A* and *B*, and acquire the ciphertext file. If the adversary tries to convert the binary ciphertext to an image they will see an image depicted in Fig. 5. We used an online raw pixel viewer tool to create the ciphertext image.

Fig. 5. Binary ciphertext file interpreted by the raw pixel viewer.

D. Quantum Decryption Procedure

For the decryption to be successful, the decrypting party *B* creates an Inverse Permutation Pad. The procedure is identical to the one described in Section *A*, however, instead of appending the permutation gates themselves to the list `Permutations`, the decrypting party *B* will append their conjugate transposes. That is, *B* will use the command

```
Permutation_matrix=np.array(([[a0,a1,a2,a3],[b0,b1,b2,b3],[c0,c1,c2,c3],[d0,d1,d2,d3]])
Inverse_Permutations.append(Operator(Permutation_matrix.transpose()))
```

to populate the list `Inverse_Permutations`. Then *B* will use the shared secret seed to create the `Inverse_Permutation_Pad` using the same technique as shown in Section *A*. That is, *B* uses the same universal way to populate the list `Inverse_Permutations` as *A* used to populate the list `Permutations`. The lists `Inverse_Permutations` and `Permutations` is simply a list of all possible 24 permutations. It is not secret and can be easily generated by anyone. The secret is the `Permutation_Pad` and `Inverse_Permutation_Pad` consisting of permutations selected from the list `Permutations` and `Inverse_Permutations` respectively in a certain order. It is this order of choosing the permutations for the Permutation Pad that is the secret. In this case, it's done using the secret seed.

Once the Inverse Permutation Pad is generated and party *B* has received the binary ciphertext file, they can start the decryption procedure. The decryption process is very similar to the encryption process and will also be performed one 2-bit block at a time using the "for" loop.

Party *B* starts by opening the binary file and extracting the ciphertext binary string. The said string is then broken into blocks of 2-bits each. Such blocks are decrypted one by one. Given a single block, party B creates a quantum circuit with 2 qubits and 2 classical bits and initializes the qubits according to the ciphertext bits in the given block. The ciphertext qubits are then acted on with an operator from the Inverse Permutation Pad. That successfully transforms the ciphertext qubits into the plaintext qubits. The plaintext qubits are then measured, and the highest probability result is stored in a list. Such a list constitutes the binary plaintext. The circuit is transplied to match the topology of the chosen quantum system. The last step for party B is to transform the binary plaintext string into a PNG file.

In Fig. 6 we depict a sample transpiled circuit for a ciphertext block at the position $x = 96$ and provide a sample measurement result of the decrypted ciphertext for the block at the position $x = 96$ in Fig. 7. We provide the code for the decrypting procedure in Fig. 8.

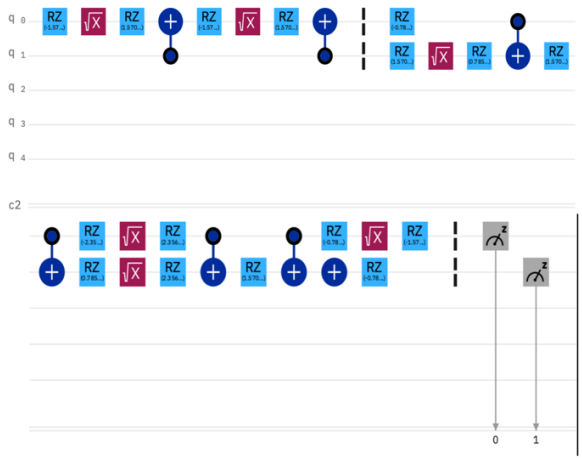

Fig. 6. Sample transpiled decryption circuit for the ciphertext block positioned at x = 96.

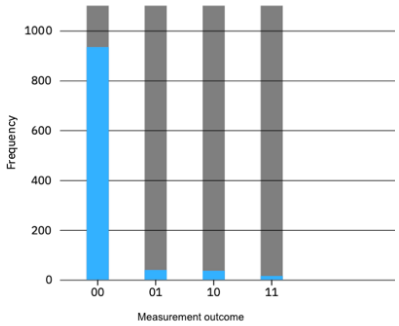

Fig. 7. Measurement result of the decrypted ciphertext block at the position x = 96. That is, the plaintext block at the position x = 96.

```
# open the ciphertxt binary file to extract the ciphertext bits
file = open("ciphertext_to_send.bin","rb")
initial_list = []
for line in file:
    initial_list.append(str(line).replace("'","").strip("b"))
chunk_size = 2
cipher_chunks = [initial_list[0][i:i+chunk_size] for i in range(0, len(initial_list[0]), chunk_size)]


list_of_messages = []
for x in range(len(cipher_chunks)):
    qc_decrypt = QuantumCircuit(2, 2)   #generate a circuit of 2 qubits and 2 classical bits
    qc_decrypt.initialize(Statevector.from_label(cipher_chunks[x]))
#initialize the input qubits to the ciphertext taken from list_of_ciphers
    qc_decrypt.barrier()


    j = Inverse_Permutation_Pad[x%len(Inverse_Permutation_Pad)]
    qc_decrypt.append(Inverse_Permutations[j], range(2))
    qc_decrypt.barrier()
    qc_decrypt.measure([0,1], [0,1])    #measure the result

    qc_decrypt = transpile(qc_decrypt, basis_gates=["u3","u2","u1","cx","id","u0","u","p","x","y","z","h","s","sdg","t","tdg","rx","ry","rz","sx","sxdg","cz","cy","swap","ch","ccx","cswap","crx","cry","crz","cu1","cp","cu3","csx","cu","rxx","rzz","rccx","rc3x","c3x","c3sqrtx","c4x"], optimization_level = 3)


#execute the circuit and store the final state in the list of messages
```

```
    provider = IBMQ.load_account()
    qcomp = provider.get_backend('ibmq_manila')
    job = execute(qc_decrypt, backend=qcomp, shots = 1024)
    job_monitor(job)
    result = job.result()
    counts = result.get_counts()
    list_of_messages.append(counts.most_frequent())   #will only
append results with highest probability

cat_decrypted = "".join(list_of_messages)
cat_decrypted_bytes = [int(cat_decrypted[i:i + 8], 2) for i in
range(0, len(cat_decrypted), 8)]

# create a PNG image with the plaintext bits
with open('decrypted_cat.png', 'wb') as f:
    f.write(bytes(cat_decrypted_bytes))z
```

Fig. 8. The source code for the decryption procedure.

We invite the reader to try and run the entire code for both encryption and decryption to verify that, indeed, the 'decrypted_cat.png' is the Schrödinger's cat image from Fig. 1.

E. Classical Decryption Procedure

Note that the ciphertext is transferred to party *B* as a binary file, due to the lack of a quantum channel. Since the ciphertext qubits can be measured to produce binary string and since QPP can be implemented in a classical computer [15, 18], the decryption can be done on a classical device. In a classical environment, the ciphertext binary file will be opened and the binary ciphertext string will be extracted. Such string will be broken into blocks of 2 bits and the state vector of each block will be multiplied by a permutation from a classically created transposed permutation pad. That would produce a plaintext that can be transformed into a PNG file.

IV. Discussion

In this section, we demonstrated quantum encryption and decryption with QPP consisting of 28 2-qubit permutation gates, yielding 128 bits of information entropy.

We point out that to achieve 128 bits of entropy, the pre-shared secret key should be at least 128 bits in length to avoid brute force search attack and should be truly random. However, the implementation described in this paper is a simplified example and is aimed to analyze whether QPP can be implemented in quantum systems. So, the key used is not 128 bits long. In general, the pre-share secret is used to generate the QPP pad. In this case, the secret would be used to generate a QPP pad with 28 of 2-qubit permutation gates expressed as matrices. This secret key must consist of $28 \times 8 = 224$ bits of random numbers, since it takes 8 bits to generate a single permutation operator from the pre-shared secret material.

The implementation described in this work can be extended to any n-qubit permutations, however, the current quantum computers are too noisy to successfully produce correct results that can be properly interpreted. Moreover, due to lack of quantum channel, the cipher qubits are measured and converted to classical bits by considering the highest probability result counts. We run the experiment anywhere from 1024 to 20,000 times. The current noisy qubits and quantum gates might produce false measurements if the circuit is only run once. Recall that process for encryption and decryption described in this work is simplified to show how the quantum encryption and decryption with QPP could be done in today's public free of charge IBMQ quantum computers. Due to the limitations of those quantum computers, we have omitted the pre-randomizing and random dispatching procedures as what we have implemented in our classical implementations in [20]-[22]. Moreover, the classical implementation circuit is more involved, as it uses the cipher chaining. We are currently working on optimizing and extending the work described in this paper to advance this implementation and match with the classical implementation reported in [20]-[22].

Nevertheless, even with the simplified implementation, the image in Fig. 1 was quantum mechanically encrypted into cipher qubits with measured cipher bits (see Fig. 5). Note that the essence of quantum encryption is using quantum gates to encrypt qubits that correspond to a binary plaintext. In this implementation we have preserved this structure and, thus, this implementation of QPP falls under what we define to be quantum encryption.

V. Conclusion

We present a functioning implementation of Kuang *et al.'s* Quantum Permutation Pad implemented using the Qiskit development tool on the currently publicly available IBM quantum computers. QPP is a symmetric encryption scheme, which we used to encrypt a PNG image on an IBM's quantum computer, and due to the lack of available publicly accessible quantum channel, we measured the ciphertext qubits and generated a binary file containing the ciphertext. Said binary file is sent to the decrypting party. The decrypting party uses the QPP inverse to decrypt the ciphertext. The implementation discussed in this paper provides 128 bits of entropy by generating a permutation pad with 28 2-qubit permutation gates. That is, each permutation matrix is acting on 2 qubits at a time. Both, the plaintext and the ciphertext are broken into blocks of 2-bits, each such block is encrypted and decrypted respectively one at a time. We inform the reader that the implementation of QPP given in this paper is not final. In fact, this paper just demonstrates an example of how to perform quantum encryption within quantum computers with QPP. To avoid statistical analysis attacks the plaintext should be randomized. Moreover, there are far better ways to generate the Permutation Pad. We are currently working on advancing the implementation of QPP to include the mentioned above properties.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Both authors contributed to the work described in this paper; The authors jointly drafted and reviewed the manuscript and approved the submission; Dr. Kuang majorly contributed on the development and the advancement of the QPP algorithm; Mrs. Perepechaenko contributed on the implementation of QPP in IBMQ systems described in this paper; all authors had approved the final version.

REFERENCES

[1] A. I. Nurhadi and N. R. Syambas, "Quantum Key Distribution (QKD) protocols: A survey," in *Proc. 4th International Conference on Wireless and Telematics (ICWT)*, 2018, pp. 1-5.

[2] E. Diamanti, H. K. Lo, B. Qi, and Z. Yuan, "Practical challenges in quantum key distribution," *Npj Quantum Inf* **,** vol. 2, 2016.

[3] O. Amer, V. Garg, and W. O. Krawec, "An introduction to practical quantum key distribution," *IEEE Aerospace and Electronic Systems Magazine*, vol. 36, no. 3, pp. 30-55, 2021.

[4] L. J. Wang, K. Y. Zhang, J. Y. Wang, *et al.*, "Experimental authentication of quantum key distribution with post-quantum cryptography," *Npj Quantum Inf* **,** vol. 7, 2021.

[5] D. Moody, G. Alagic, D. Apon, *et al*, "Status report on the second round of the NIST post-quantum cryptography standardization process," NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Unpublished, 2020.

[6] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, pp. 710-722, 2003.

[7] P. W. Shor, "Polynomial-Time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, pp. 1484-1509, 1997.

[8] L. K. Grover, "A fast quantum mechanical algorithm for database search," *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pp. 212-219, 1996.

[9] X. Bonnetain, M. Naya-Plasencia, and A. Schrottenloher, "Quantum security analysis of AES," *IACR Transactions on Symmetric Cryptology*, pp. 55-93, 2019.

[10] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying grover's algorithm to AES: Quantum resource estimates," in *Post-Quantum Cryptography*, 2016, pp. 29–43.

[11] P. Kim, D. Han, and K. C. Jeong, "Time–space complexity of quantum search algorithms in symmetric cryptanalysis: Applying to AES and SHA-2," *Quantum Information Processing*, vol. 17, pp. 1–39, 2018.

[12] B. Langenberg, H. Pham, and R. Steinwandt, "Reducing the cost of implementing the advanced encryption standard as a quantum circuit," *IEEE Transactions on Quantum Engineering*, vol. 1, pp. 1–12, 2020.

[13] Z. Wang, S. Wei, and G. Long, "A quantum circuit design of AES requiring fewer quantum qubits and gate operations," unpublished, 2021.

[14] J. Zou, Z. Wei, S. Sun, X. Liu, and W. Wu, "Quantum circuit implementations of AES with fewer qubits," *Advances in Cryptology – ASIACRYPT*, pp. 697-726, 2020.

[15] R. Kuang and N. Bettenburg, "Shannon perfect secrecy in a discrete hilbert space," in *Proc. IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2020, pp. 249-255.

[16] A. Ambainis and A. Smith, "Small pseudo-random families of gates: Derandomizing approximate quantum encryption", *Lecture Notes in Computer Science*, vol. 3122, 2004.

[17] P. O. Boykin and V. Roychowdhury, "Optimal encryption of quantum bits," *Physical Review A*, vol. 67, 2003.

[18] R. Kuang and M. Barbeau, "Quantum permutation pad for universal quantum safe cryptography," *Quantum Inf Process*, 2022.

[19] M. Perepechaenko and R. Kuang, "Quantum encrypted communication between two IBMQ systems using quantum permutation pad," in *Proc. 11th International Conference on Communications, Circuits and Systems (ICCCAS)*, 2022, pp. 146-152.

[20] R. Kuang, D. Lou, A. He, and A. Conlon, "Quantum safe lightweight cryptography with quantum permutation pad," in *Proc. IEEE 6th International Conference on Computer and Communication Systems*, 2021, pp. 790-795.

[21] R. Kuang, D. Lou, A. He, C. McKenzie, and M. Redding, "Pseudo Quantum Random Number Generator with Quantum Permutation Pad," in *Proc. IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2021, pp. 359-364.

[22] D. Lou, R. Kuang, and A. He, "Entropy transformation and expansion with quantum permutation pad for 5G secure networks," in *Proc. IEEE 21st International Conference on Communication Technology (ICCT)*, 2021, pp. 840-845.

Maria Perepechaenko received a B.S in mathematics from University of Toronto in 2018 and a M.Sc in mathematics from University of Ottawa in 2021. During her studies, her main research interest was post-quantum cryptography and the Hidden Subgroup Problem, closely related to lattice-based cryptography. Currently, she works at Quantropi Inc. as an R&D Associate. In this context, she participates in the development, analysis, and advancement of Quantropi's symmetric and asymmetric cryptographic algorithms, with work published in the IEEE Xplore, and Scientific Reports.

Dr. Randy Kuang received his BSc in Physics from Sichuan Normal University, China, in 1983, Msc in Atomic and Molecular Physics from Sichuan University, China, in 1986 and PhD in Atomic and Molecular Physics from Memorial University of Newfoundland, Canada, in 1996. His breakthrough research and findings have been published in prestigious journals around the world, and his innovative united atom model in 1991 was even coined "Kuang's semi-classical formalism" by NASA

in 2012. His professional career includes stops at Nortel Networks as a senior researcher, inBay Technologies as Co-Founder and CTO, and most recently, Co-Founder and Chief Scientist at Quantropi, a quantum-secure communications company. Randy is a prolific inventor with 39 U.S. patents under his belt in broad technology fields such as WiMAX, optical networks, multi-factor identity authentication, quantum cryptography, and post-quantum cryptography. Specifically, Randy invented two-level authentication (2009) leading to today's authentications with smart phones, quantum permutation pad for quantum encryptions (2018, 2019) paving the way for quantum secure communications over today's Internet and future's quantum internet, quantum public key distribution with randomized coherent states (2019) for wire speed key distributions and data communications over today's coherent optical networks, and quantum safe multivariate polynomial public key (2020, 2022) for key exchange mechanism and digital signature.