Kuang and Perepechaenko *EPJ Quantum Technology* (2022) 9:26
https://doi.org/10.1140/epjqt/s40507-022-00145-y

EPJ Quantum Technology
a SpringerOpen Journal

RESEARCH

Open Access

Quantum encryption with quantum permutation pad in IBMQ systems

Randy Kuang[1*] and Maria Perepechaenko[1]

*Correspondence:
randy.kuang@quantropi.com
[1]Quantropi Inc., Ottawa, Canada

Abstract

Quantum permutation pad or QPP is a quantum-safe symmetric cryptographic algorithm proposed by Kuang and Bettenburg in 2020. The theoretical foundation of QPP leverages the linear algebraic representations of quantum gates which makes QPP realizable in both, quantum and classical systems. By applying the QPP with 64 of 8-bit permutation gates, holding respective entropy of over 100,000 bits, we accomplished quantum random number distributions digitally over today's classical internet. The QPP has also been used to create pseudo quantum random numbers and served as a foundation for quantum-safe lightweight block and streaming ciphers. This paper continues to explore numerous applications of QPP, namely, we present an implementation of QPP as a quantum encryption circuit on today's still noisy quantum computers. With the publicly available 5-qubit IBMQ devices, we demonstrate quantum secure encryption (256 bits of entropy) using 2-qubit QPP with 56 permutation gates, and 3-qubit QPP with 17 permutation gates respectively. Initial qubits of the encryption circuit correspond to the plaintext and after applying quantum encryption operations, cipher qubits are measured with probabilistic distributions, and the results with the highest probability are recorded as cipher bits. The cipher bits are then decrypted with an inverse QPP circuit. The output state plaintext qubits are measured and the most frequent count measurement results are recorded as plaintext bits. This quantum encryption and decryption process clearly demonstrates that QPP quantum implementations works exactly as symmetric encryption and decryption schemes should. The plaintext and ciphertext bits can also be encrypted and decrypted respectively by any classical computing device with the corresponding QPP algorithm as in quantum computers. This work reveals that it is possible to build quantum-secure communications between quantum-to-quantum and quantum-to-classical computers over today's internet and the future quantum internet.

Keywords: Quantum permutation gate; Quantum encryption; Quantum decryption; Quantum circuits; Qiskit; Symmetric encryption; QKD; Symmetric cryptography; QPP; Quantum communication

1 Introduction

The term "Quantum encryption" often refers to quantum cryptography or Quantum Key Distribution (QKD) used to establish a secret key for digital symmetric encryption with quantum-resistant algorithms such as Advanced Encryption Standard (AES) and One-Time-Pad (OTP). The QKD was first proposed by Bennett and Brassard in 1984 [1], leveraging the physical uncertainty of quantum measurement in conjugate encoding bases. In the past few decades several variants of QKD have been developed. Among the said variants, some are based on the single photon 2-party protocol called the discrete variable QKD or DV-QKD described in Djordjevic's book [2], as well as some recent papers by Lai *et al.* [3] and Qi [4]. Others are based on the 2-party continuous variable QKD using coherent states or CV-QKD found in the works of Stefano *et al.* [5–7]. Both, DV-QKD and CV-QKD suffer from the key rate upper bound due to the distance limitations. In attempt to overcome the key rate-distance constraint, a new three-party protocol called the twin-field QKD or TF-QKD was proposed by Lucamarini *et al.* in 2018 [8]. In 2021, Chen *et al.* extended the attainable distance of the TF-QKD to over 500 km [9]. This result was further improved by Wang *et al.* in 2022 [10] to over 800 km. With the establishment of shared secrets using QKD, two communicating parties can achieve quantum secure communication by employing symmetric encryption algorithms such as AES with 256-bit key. Therefore, the said quantum encryption consists of two separate processes: quantum key distribution using QKD and classical data encryption using symmetric algorithms.

Quantum secure direct communication without pre-shared key or QSDC was proposed by Deng, Long and Liu in 2003 [11]. The QSDC uses blocks of EPR pairs. A set of ordered EPR pairs is split into two sequences: checking and message-coding. Checking sequence is used to verify the channel integrity and establish encoding and measuring bases between the sender and the receiver. After the checking phase, a message can be encoded based on the bases determined in the checking phase. The encoded message is then transmitted over the quantum channel and finally measured at the receiving end in the bases determined in the checking phase. The QSDC can be considered a combination of key distribution and message encryption, both done quantum mechanically. Deng *et al.* in 2004 further advanced the QSDC and proposed a new QSDC protocol with a batch of single photons as a quantum OTP [12]. Zhang *et al.* in 2017 proposed their QSDC with quantum memory [13].

As the development of universal fault-tolerant quantum computers showed significant speedup in the past few years, quantum encryption with quantum circuits have been proposed based on different encryption algorithms. Almazrooie *et al.* in 2018 proposed their quantum circuit design of AES-128, requiring the maximum of 928 qubits. Langenberg *et al.* in 2020 proposed an improved design requiring a total of 880 qubits, 1507 X gates, 107960 CNOT gates, and 16940 Toffoli gates [14]. Wang, Wei and Long in 2021 further reduced the implementation to 656 qubits, 1976 X gates, 101174 CNOT gates, but 18040 Toffoli gates (higher compared to the design of Langenberg *et al.*) [15]. Zou *et al.* in 2020 proposed their implementation of AES quantum circuit design with some optimization considerations on AES S-Box and key scheduling. The proposed implementation requires 512 qubits for AES-128 [16]. Some other lightweight symmetric encryption algorithms have been designed for quantum circuits such as PRESENT and GIFT by Jang *et al.* in 2021 [17] and RECTANGLE and KNOT by Baksi *et al.* in 2021 [18].

Possible implementations of the described quantum circuit designs for the classical symmetric encryption algorithms involve considerably large quantum resources. It is clear that efficient implementation of said quantum algorithms will have to wait for relatively long time. Hu and Kais in 2021 proposed a lightweight quantum encryption scheme [19], using a generic unitary gate with N discrete probabilistic amplitudes per qubit for a block of n-qubits. To each qubit of an n-qubit block a unitary gate is applied, followed by a set of CNOT gates to enforce diffusion and confusion capability to cipher quantum states. The cipher quantum states are generally in superposition, thus preventing possible eavesdropping during their transmission to a receiver. They also proposed two modes of operations to enhance the security. Such design would work for secure quantum communications between two quantum computers given an ideal quantum channel.

Leveraging the entire Symmetric group of permutations with $2^n!$ elements, Kuang filed his first patent in 2017 to perform encryption with permutation matrices [20] then filed its enhancement in 2019 [21]. Kuang and Bettenburg in 2020 formally proposed an encryption algorithm using a pad of permutations called the Quantum Permutation Pad or QPP [22]. It is well-known that permutation transformations are bijective over computational basis, so the QPP encryption holds the property of Shannon perfect secrecy. The QPP encryption scheme has been applied to different cases recently in classical computing [23–25]. For an n-qubit computational basis, there exist a total of $2^n!$ unique permutation gates. The entire group of permutations corresponds to the Shannon information entropy of $\log_2(2^n!)$ bits. Considering this permutation group as a key space and mapping the classical key material into a QPP pad by using random shuffling algorithm such as the Fisher-Yates algorithm, a new cryptographic algorithm can be established that is applicable to both classical computing and quantum computing.

In this paper, we propose an implementation of QPP in fully available IBM quantum computers. Due to its relative simplicity, we could create a hybrid quantum-classical scheme of quantum encrypted secure communications over a potential hybrid internet. While this paper focuses entirely on the implementation of QPP in quantum systems, detailed security analysis can be found in a very recent publication by Kuang and Barbeau in 2022 [26].

2 Quantum permutation pad

We refer to *quantum encryption* to describe an encryption scheme that employs unitary operators for encryption and their respective Hermitian conjugate operators for decryption. That is, if such unitary operators were constructed using a pre-shared secret key then both parties can perform encryption and decryption using said unitary operators, and the adversary will have to obtain the pre-shared secret key in order to determine such operators. One great example of such operators are permutation operators. In the classical settings, permutation operators are elements of the symmetric group S_{2^n} which permute the 2^n elements $\{0,\dots,2^{n-1}\}$, or equivalently, $2^n \times 2^n$ matrices. There are $2^n!$ such permutation operators. As matrices, permutation operators are natural quantum computing objects. That is, permutation operators can be constructed directly using the matrix form of the permutations. Moreover, it can be easily shown that permutation operators are indeed unitary operators.

We want to warn the reader that the quantum permutation operators used in quantum computing are very different from the notion of a permutation operator in quantum physics, where permutation operators act on identical particles and mainly refer to swapping particles' physical positions. In quantum physics, it is known that the total number of permutations of n identical particles is $n!$. In quantum computing, on the other hand, permutation operators act as quantum perturbations of an n-qubit system and permute its computational basis from $B_c = \{|0\rangle, |1\rangle, |2\rangle, \dots, |2^n-1\rangle\}$ to $B_P =$

$\{\hat{P}|0\rangle, \hat{P}|1\rangle, \hat{P}|2\rangle, \ldots, \hat{P}|2^n - 1\rangle\}$. It is clearly seen that B_P is just a permuted basis of the computational basis B_c. There are in total $2^n!$ permuted bases for an n-qubit system.

The entire set of quantum permutation gates forms the symmetric group S_{2^n} or so-called Quantum Permutation Space or QPS. QPS can be considered as "quantum key space" with a dimension $2^n!$, which is significantly larger than its corresponding classical key space with a dimension of 2^n. Such dramatic increase in the dimension of the key space from classical computing to quantum computing indicates the exponential increase of quantum key entropy to be used for quantum encryption. Therefore, the paradigm shift of computing algebra from Boolean to linear not only reveals the superpower of quantum computing but also releases the superpower of entropy for quantum encryption.

Kuang et al. reported their QPP implementations as a lightweight quantum safe block cipher [23] and streaming cipher [24], entropy transformation and expansion [25], pseudo quantum random number generation [27]. Kuang and Barbeau recently proposed a universal quantum safe cryptography with QPP [26] for potential quantum encrypted communications between two quantum computers and one quantum one classical computers over the existing internet or future quantum internet. Perepechaenko and Kuang have demonstrated their first toy implementation of QPP in IBM 5-qubit quantum computer recently [28]. In this paper, we describe an implementation of QPP on today's publicly-available free of charge quantum computers such as `ibmq-bogota` to demonstrate how we can achieve quantum encryption and decryption with at least 256 bits of entropy, even with today's noisy quantum computer.

For brief summarization, we will introduce 1-qubit QPP, 2-qubit QPP and n-qubit QPP in the following subsections separately. The QPP can be designed and described using two different mechanisms. On one hand, QPP can be expressed using eigen-decomposition of the permutation operators. Such QPP gets physical security from the uncertainty principle. On the other hand, QPP can be considered using the permutation matrices themselves. This QPP gets its information-theoretical security from uninterpretability of the states that have been acted on with permutation operators. We discuss the latter in the next few sections, as our implementation only uses the permutation matrix interpretation of QPP.

2.1 1-qubit QPP

A single qubit system, can be in only two possible states. Thus there are only $2^1! = 2$ permutation matrices acting on such states, namely the identity $P_0 = I = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ and $P_1 = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ also known as a NOT gate. The computational basis of such 1-qubit system is $B_c = \{|0\rangle, |1\rangle\}$, where $|0\rangle = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$ and $|1\rangle = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$. Suppose that the system is prepared in the state $|a\rangle$, where $a \in \{0, 1\}$. The identity permutation P_0 applied to the said state does not change the computational basis. Indeed, $B_{P_0} = \{\hat{P}_0|0\rangle, \hat{P}_0|1\rangle\} = \{|0\rangle, |1\rangle\}$. On the other hand, P_2 would change state $|0\rangle$ to $|1\rangle$ and state $|1\rangle$ to $|0\rangle$. Thus, $B_{P_1} = \{\hat{P}_1|0\rangle, \hat{P}_1|1\rangle\} = \{|1\rangle, |0\rangle\}$. A single qubit state can be still deterministically measured in the computational basis after a permutation but it can only be probabilistically interpreted without knowing what the permutation operator is operated. That is, if the permutation is randomly chosen from $\{P_0, P_1\}$ and applied to the state $|a\rangle$, and the adversary is only able to see the resulting state, they will not be able to determine if the original state is $|a\rangle = |0\rangle$ or $|a\rangle = |1\rangle$. Thus, the correct interpretation of the original state, without any other knowledge, happens only 50% of the time.

In single qubit case, a random key pad can be naturally mapped to a quantum permutation pad or QPP with key bit '0' for P_0 and '1' for P_1. Then the implantation circuit has two qubits, one for the secret key, and one for the qubit to be acted on. Quantum implementation of the permutations P_0 and P_1 can be done using a 2-qubits $Control-NOT$ or CNOT gate, with key bit used as control qubit. The output of said operation is a secret key qubit and the cipher qubit. It is clearly seen that CNOT gate performs the classical XOR bitwise operation. Therefore, we can implement the classical OTP encryption with quantum logic gate CNOTs with each pair of key qubit and message qubit as input qubits for the CNOT gate.

The decryption is done the same way, however, using the $P_0^\dagger$ and $P_1^\dagger$, and a cipher qubit instead of the message qubit. In the case of 1-qubit QPP implementation $P_0^\dagger = P_0$ and $P_1^\dagger = P_1$.

Although CNOT gates operate on qubits, the quantum encryption with CNOT gates behaves the exactly same way as the classical OTP, that means, the key qubits can only be used for one time because of the deterministic measurements in the computational basis. In order to make a key reusable, we may have to use either superposition single qubit gates such as a universal gate [19] or Hadamard or H gate.

2.2 2-qubit QPP

For 2-qubit permutation space, there is a total of $2^2! = 24$ permutation gates. Indeed, there are 4 possible states of the system with 2 qubits, and we apply permutation operator to the entire system, thus applying to 2 qubits simultaneously. Here is a typical 2-qubit permutation operator in its matrix form

$$P_0 = \text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Such permutation maps the state $|2\rangle = |10\rangle_b$ (note: we use subscript b to a state vector for bit string) to $|3\rangle = |11\rangle_b$ and $|3\rangle = |11\rangle_b$ to $|2\rangle = |10\rangle_b$, and leaves the states $|0\rangle = |00\rangle_b$ and $|1\rangle = |01\rangle_b$ intact. Another example of the 2-qubits permutation operator is

$$P_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Such permutation turns the state $|0\rangle = |00\rangle_b$ to $|1\rangle = |01\rangle_b$, $|1\rangle$ to $|3\rangle = |11\rangle_b$, $|2\rangle = |10\rangle_b$ to $|0\rangle$, and $|3\rangle$ to $|2\rangle$. Suppose that the message qubits are $|3\rangle = |11\rangle_b$ and the secret key determined that P_1 is the permutation that will be used for encryption, then the produced ciphertext state is $|2\rangle = |10\rangle_b$. The decryption can be done in the same way using conjugate transpose of P_1 applied to the ciphertext state to produce the message state.

We can randomly select a pre-shared 2-qubit QPP pad, from 2-qubit permutation space, to perform direct quantum encryption for uninterpretable security or its eigen-decomposition pad for physical untouchable security. However, quantum implementation with an eigen-decomposition QPP pad or 2-qubit QKD is much more complex than using a single qubit QPP pad. Note that 2-qubit QKD with eigen-decompisitions is not likely practical.

2.3 n-qubit QPP

Although quantum encryption in permutation eigenbases makes the encryption key reusable because of the physical uncertainty principle, the encryption can only work within quantum computing systems or between quantum computers over an ideal quantum channel. In order for a key to be reused for quantum encryption with a QPP pad, we have another option: using n-qubit permutation gates with $n > 1$ because of the generalized uncertainty principle or mathematical non-commutativity: $[\hat{P}_i, \hat{P}_j] \neq 0$ for $i \neq j = 1, 2, \ldots, 2^n!$. This is fundamental for QPP cryptographic algorithm to not only hold the property of Shannon perfect secrecy with uninterpretable security but also make the key reusable.

Quantum encryption with QPP generally has following steps:

- The first step is to decide how many permutation gates are required to achieve at least 256 bits of entropy for a given n-qubit permutation space. Table 1 illustrates the dimension of different permutation spaces, equivalent Shannon entropy per permutation gate, number of permutation gates for a QPP to achieve at least 256 bits of entropy, length of classical key materials required, and number of qubits for a QPP encryption circuit. To achieve quantum security, we need the randomly chosen QPP pad with at least 56 gates for using 2-qubit permutation space, or 17 gates for using 3-qubit permutation space, or 6 gates for using 4-qubit permutation space, or 3 gates for using 5-qubit permutation space.
- The second step is to decide the classical key length. Table 1 also displays the classical key lengths required to produce the QPP pad. Generally speaking, the minimum random key length per permutation gate requires ($\log_2 2^n!$) bits and the maximum key length is $n \times 2^n$ bits. Table 1 illustrates the maximum classical key length for 2-qubit, 3-qubit, 4-qubit, and 5-qubit per QPP pad, respectively. They all hold more than 256 bits of entropy. However, their corresponding QPP circuits require much less than 256 qubits, thanks to the condensed entropy from the permutation space.
- The third step is to map the classical key materials into a QPP pad. There are a number of algorithms to chosen for this mapping. We choose the Fisher-Yates shuffling algorithm to shuffle the states of the n-bit finite field. For a sufficient shuffling, a random bit string should be $n \times 2^n$ bits to choose one permutation gate.

Table 1 The table tabulates the equivalent Shannon information entropy per n-qubit permutation space for n from 2 to 5. We also illustrate the number of gates to achieve the equivalent classical 256 bits of entropy and classical key bit length. The last row displays the number of qubits required per circuit to achieve the quantum security, counting qubits corresponding to the message and the pre-shared key

	2-qubits	3-qubits	4-qubits	5-qubits
Dimension of Permutation Space	24	40,320	2.09×10^{13}	2.63×10^{35}
Entropy per Permutation Gate (bit)	4.58	15.30	44.25	117.7
Number of Permutation Gates Required	56	17	6	3
Classical key length (bit)	448	408	384	480
Total Qubits Required	224	102	48	30

- the forth step is to decide how many qubits a QPP circuit needs for NIST security level V or 256 bits of entropy. Table 1 displays that a QPP circuit requires 112 qubits for a 2-qubit pad, or 51 qubits for 3-qubit pad, or 24 qubits for 4-qubit pad, or 15 qubit for 5-qubit pad. It is obvious that full implementation of a QPP circuit in today's quantum computer faces challenges due to the limitation of quantum volumn. However, we can demonstrate the fundamental logics with certain simplifications to be discussed later.

3 QPP circuits

Any implementation of quantum encryption using QPP should consist of three main building blocks, namely the randomization operation, the dispatching operation, and the permutation operation. The randomization operation is to be applied to the plaintext to randomize it and erase any statistical patterns that can be later used for statistical analysis attacks. The dispatching operation determines which permutation operator of the Permutation Pad will be applied to a given n-qubit randomized plaintext state. The permutation operation describes the encryption of a given randomized plaintext using the dispatched permutation operators.

In this section, we give an overview of an ideal implementation of QPP on future fault-tolerant, fully scalable quantum computers with sufficient quantum volume. Such implementation should be done using tools entirely native to quantum computing. For instance, the randomization and the dispatching operations should be implemented using quantum gates only. The quantum Permutation Pad itself ought to be created using exclusively quantum gates as well. We illustrate a sample circuit for this implementation in Fig. 1.

An attentive reader will notice that in Fig. 1 the initialization operator is depicted using a dashed line rather than a solid line. That is because the default initial state depends on

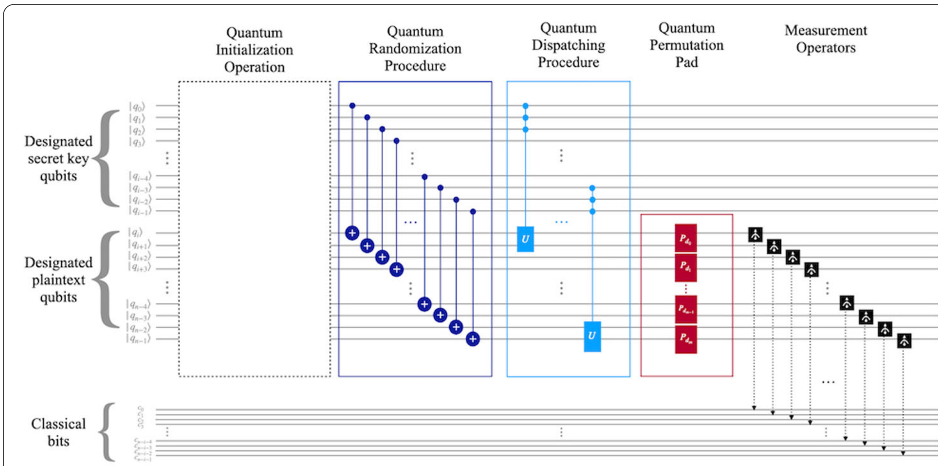

Figure 1 We illustrate an encryption circuit for ideal implementation of the QPP on a quantum device. All the operations are performed using quantum gates and instruments native to quantum computing. The encryption procedure consists of an initialization, randomization, dispatching and quantum permutation pad operations. Note that permutations in the Permutation Pad are applied together as one pad. For this specific figure each permutation operator is acting on 2 qubits at a time. The permutation operators are not limited to 2 qubits only, but for the sake of simplicity without loss of generality we depict 2-qubits permutation operators. Qubits labeled "Designated plaintext qubits" and "Designated secret key qubits" are denoted in such way to show what input they will store. Note, that they are not yet initialized according to the plaintext bits and secret key bits. Initialization operation will transform default qubits into precisely the secret key qubits and plaintext qubits. That is, such qubits if measured will be precisely the plaintext and secret key bits

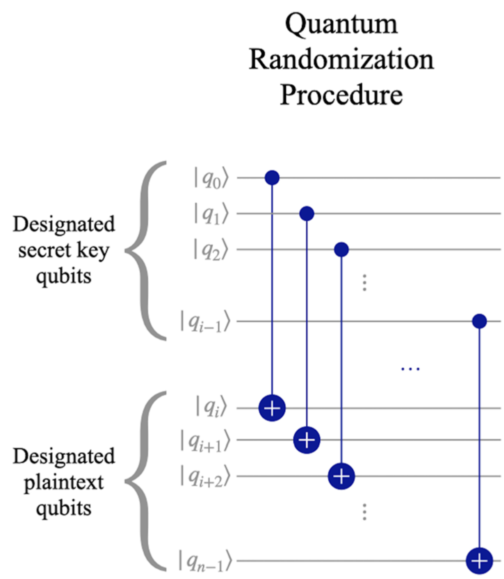

Figure 2 Quantum Randomization procedure is a collection of CNOT gates applied to the plaintext qubits with secret key qubits set as control. Such implementation is the quantum counterpart of the classical XOR operation. Indeed, suppose the secret key is $011\ldots1$ and the plaintext is $111\ldots0$. The result of the XOR operation between the said two bit strings is $100\ldots1$. Setting the control qubits as depicted in this figure for $q_0 = 0, q_1 = 1, q_2 = 1, \ldots, q_{i-1} = 1$ and $q_i = q_{i+1} = q_{i+2} = 1, \ldots, q_{n-1} = 0$ will produce the state $|100\ldots1\rangle$, which when measured is precisely the bit-string $100\ldots1$

the hardware. In the case of the IBM hardware, used for our implementation, each of the default qubits is set to $|0\rangle$. Thus, in order to act on the initial state of the quantum system in the same way as we would act on the binary plaintext, we need to initialize the qubits according to the plaintext bits. For different quantum hardware, the qubits might not be set to any specific default value, thus, no initialization might be required.

In the framework of a fully quantum implementation, the randomization procedure can be done by a collection of CNOT gates applied to the plaintext qubits with secret key qubits set as control qubits. We depict the described randomization operation in Fig. 2.

Dispatching can be done in many different ways. The fully quantum dispatcher will use the qubits initialized according to the pre-shared secret key bits, to determine which permutation from the Permutation Pad will be applied to any given plaintext qubits. We have not yet determined the most optimal way to implement such dispatching, however, as with the randomization operation, secret key qubits will be controlling the dispatching operator. Thus, we will illustrate it in Fig. 3 as controlled operations with key qubits set as control qubits.

In the future implementation of QPP, permutation operators are applied as a single Permutation Pad. However, each permutation in the Permutation Pad is acting on a specified number of qubits in the fashion determined by the dispatcher. That is, the first k-qubits permutation in the Permutation Pad might not be acting on the first k qubits of the randomized plaintext, but rather on the k-qubits block it was dispatched to. We denote a permutation operator that acts on k qubits simultaneously as a k-qubits permutation operator. The illustration of a sample Permutation Pad with 2-qubits permutation operators is depicted in Fig. 4.

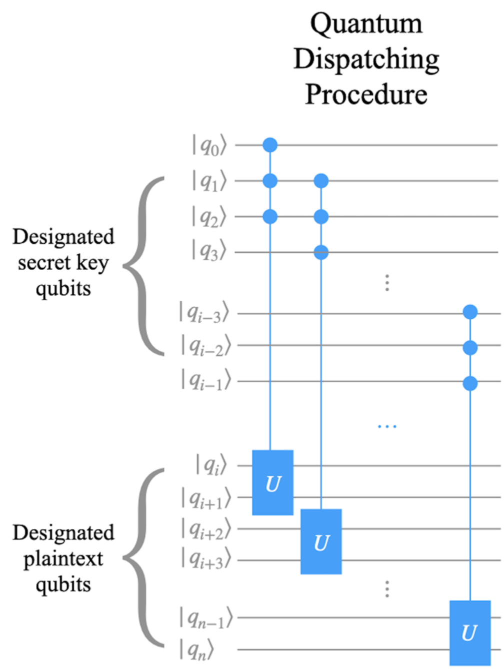

Figure 3 The dispatching procedure, in the ideal case, will be performed using quantum gates applied to the plaintext qubits with the secret key qubits set as control qubits. Note that we have not yet determined the optimal way to implement the quantum dispatching procedure, however, it should depend on blocks of secret key qubits and act of blocks of plaintext qubits. Thus, we depict dispatching as quantum gates applied to the blocks of plaintetx qubits with multiple secret key qubits set as constrol. We also believe that the quantum gates used for dispatching are unitary gates. After all, their classical counterparts are simply blocks of secret key converted to integers

The said circuit will successfully encrypt any given plaintext using quantum gates and qubits only. In the future, the encrypting party can simply send the ciphertext qubits, to the decrypting party over a reliable quantum channel.

Once the decrypting party receives the ciphertext qubits, they can begin the decryption procedure. The first step is to use the dispatcher that will dispatch operators from the Inverse Permutation Pad. Note that the two communicating parties will agree on the dispatching procedure as well as the pre-shared secret key, this will guarantee that the decrypting party is able to dispatch precisely the respective Hermitian conjugates of the permutations used by the encrypting side. In this text, we will refer to the Hermitian conjugates as inverse permutations. The dispatching operation is followed by applying the dispatched inverse permutation operators from the Inverse Permutation Pad. At this point, the system is in the state that corresponds to the randomized plaintext qubits. The last step necessary to produce the plaintext is de-randomization. Using the same key and CNOT operations as described in Fig. 2, the decrypting party can generate a state that corresponds to the original plaintext. Such a state can be measured to observe the plaintext bits. The circuit for the procedure is available in Fig. 5. Each individual operation is equivalent to the operations illustrated in Fig. 2, 3, 4, however, the plaintext qubits will be replaced by the ciphertext qubits, and the Inverse Permutation Pad consists of inverses of the permutations from the Permutation Pad.

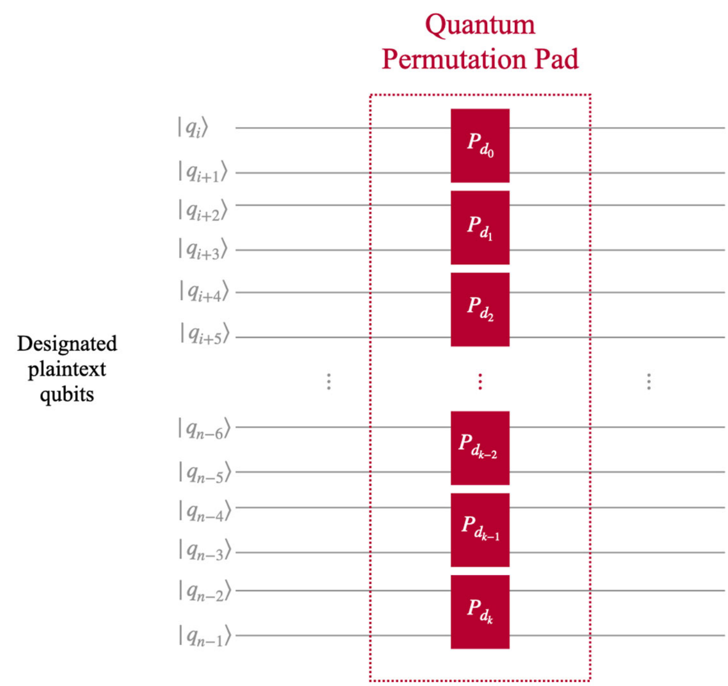

Figure 4 The Quantum Permutation Pad in the case of ideal fully-quantum permutation will be applied as one pad to plaintetx qubits. For this specific illustration each permutation in the permutation pad acts on 2 qubits simultaneously. Permutation operators are not limited to 2 qubits at a time, however, we depict it for the sake of simplicity without loss of generality using 2-qubits permutations

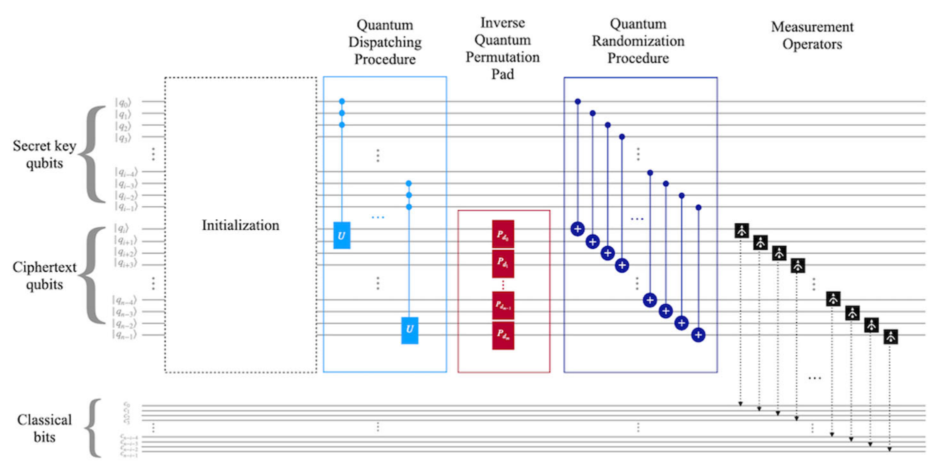

Figure 5 We illustrate a decryption circuit for the ideal implementation of the fully quantum QPP. The said circuit is similar to the one for encryption, however, the qubits are initialized to the ciphertetx bits instead of the plaintext bits, and the order in which the operations are applied is different, For instance, the randomization in the decryption circuit is the last procedure that is acting on qubits, whereas for encryption is the second one. Moreover, the permutations are not dispatched form the Permutation Pad but rather the Inverse Permutation Pad. The Inverse Permutation Pad is applied to the ciphertetx qubits all together as one pad

3.1 Entropy

Depending on the desired security, the communicating parties can agree on specific parameter values. For instance, suppose that A and B agree to communicate using QPP such that the best possible attack has complexity of at least $\mathcal{O}(2^{256})$. This means that the pre-shared secret key must be truly random and be at least 256 bits in length to avoid brute force attacks. Such pre-shared key has at least 256 bits of entropy. On the other hand, the attack on the algorithm itself should require brute force search with complexity $\mathcal{O}(2^{256})$. Thus, the Permutation Key Space, as described in [26], should have 256 bits of entropy. This is possible if the Permutation Pad consists of sufficient number of permutations. For an implementation of n-qubit QPP with n-qubit permutations, the Permutation Pad must consist of at least $k = \frac{256}{\log_2(2^n!)}$ permutation operators. Since the pre-shared key $\mathcal{K}$ is used to generate the Permutation Pad, it must now satisfy two conditions, namely $\|\mathcal{K}\| > 256$ bits, and $\|\mathcal{K}\| > k \times (2^n \times n)$. This way the pre-shared key is sufficiently long to generate k permutations for the Permutation Pad.

Recall that in the future implementation of QPP the permutation operators are applied as a single Permutation Pad. This means, every quantum circuit used for encryption and decryption of a message or a block of a message should have $k \times n$ qubits corresponding to the message, and $k \times n$ qubits corresponding to the pre-shared key.

We summarize this discussion in Table 1, which illustrates parameters needed to achieve classical entropy of 256 bits with n-qubit QPP for $n = 2, \dots, 5$.

3.2 2-qubit QPP circuit

We now describe a special case of the future QPP implementation, namely the 2-qubit case. That is, the permutation operators used in the Permutation Pad are 2-qubit permutation operators. In this case, the Permutation Pad must consist of 56 permutations as shown in Table 1 to provide 256 bits of entropy. The number of qubits per circuit required to reflect the plaintext and the pre-shared secret key is 224 as illustrated in Table 1. Note also that the pre-shared secret key must be at least 448 bits of length to avoid brute force search attacks on the secret key and generate the Permutation Pad.

Suppose that the encrypting party wants to securely send a $\mathcal{K}$-bit message. Suppose, also, that the communicating parties have pre-shared a $\mathcal{K}$-bit secret key, where $\mathcal{K} \geq 448$. Then the encryption and decryption procedures are illustrated in Algorithm 1 and Algorithm 2 respectively.

If the message needed to be encrypted is less than 112 bits, then only the necessary amount of permutations will be selected from the pad. That is, the entire Permutation Pad serves as a menu, from which only the needed amount of 2-qubit permutations will be selected using the secret key and applied to the plaintext. The amount of entropy, in this case, remains the same, however, instead of applying the entire Permutation Pad at once only a certain amount of permutations is selected from the said pad. The same holds true for decryption.

3.3 3-qubit QPP circuit

The future implementation of the 2-qubit QPP can be extended to any n-qubit permutation operators. As quantum machines advance, we expect the number of qubits that can be encrypted at a time to increase drastically as well as the number of qubits a single permutation can act on at once.

Algorithm 1 Ideal 2-qubit QPP Encryption circuit

Require: $m \leftarrow$ message, $s \leftarrow$ pre-shared secret key, $i \leftarrow 112$, $k = 56$, $List = []$.
Ensure: $c \leftarrow$ ciphertext

1: CREATE Permutation Pad(s, k, i)
  return *PermutationPad*;
2: SPLIT m
  return $mBlock\_j \ \forall j = 0, \ldots, \lceil \frac{\mathcal{K}}{112} \rceil$ s.t. $\|mBlock\_j\| = 112$ bits;
3: SPLIT s
  return $sBlock\_j \ \forall i = 0, \ldots, \lceil \frac{\mathcal{K}}{112} \rceil$ s.t. $\|sBlock\_j\| = 112$ bits;
4: **for** $j = 0, \ldots, \lceil \frac{\mathcal{K}}{112} \rceil$ **do**
5: CREATE Quantum Circuit(224 qubits, 112 bits)
6: INITIALIZE ($CONCAT(mBlock\_j, sBlock\_j)$)
   return $InitialState = CONCAT(mQubitsBlock\_j, sQubitsBlock\_j)$
7: RANDOMIZE(*InitialState*) {Fig. 2}
   return *mQubitsRandomBlock_j*
8: DISPATCH Permutations($InitialState[sQubitsBlock\_j]$, *PermutationPad*) {Fig. 3}
9: ENCRYPT(*mQubitsRandomBlock_j*) by applying dispatched operators
   return *cQubitsBlock_j*
10: APPEND *cQubitsBlock_j* to the *List*
11: **end for** {Fig. 1}
12: **return** $c \leftarrow List$

Algorithm 2 Ideal 2-qubit QPP Decryption circuit

Require: $c \leftarrow$ ciphertext, $s \leftarrow$ pre-shared secret key, $i \leftarrow 112$, $k = 56$, $List = []$.
Ensure: $m \leftarrow$ message

1: CREATE Permutation Pad(s, k, n)
  return *PermutationPad*;
2: SPLIT s
  return $sBlock\_i \ \forall j = 0, \ldots, \lceil \frac{\mathcal{K}}{112} \rceil$ s.t. $\|sBlock\_j\| = 112$ bits;
3: **for** $i = 0, \ldots, \lceil \frac{\mathcal{K}}{112} \rceil$ **do**
4: CREATE Quantum Circuit(224 qubits, 112 bits)
5: INITIALIZE ($CONCAT(cBlock\_j, sBlock\_j)$)
   return $InitialState = CONCAT(cQubitsBlock\_j, sQubitsBlock\_j)$
6: DISPATCH Permutations($InitialState[sQubitsBlock\_j]$, *PermutationPad*)
7: DECRYPT(*cQubitsBlock_j*) by applying dispatched operators
   return *mQubitsRandomBlock_j*
8: DE-RANDOMIZE(*mQubitsRandomBlock_j*, $InitialState[sQubitsBlock\_j]$)
   return *mQubitsBlock_j*
9: MEASURE(*mQubitsBlock_j*)
   return *mBlock_j*
10: APPEND *mBlock_j* to the *List*
11: **end for** {Fig. 5}
12: **return** $m \leftarrow List$

For the 3-qubit QPP, the same $\mathcal{K}$-bit message can be broken down into blocks of 51 bits each. Then each such block can be converted into qubits and encrypted using a circuit similar to the 2-qubit case. The pre-shared $\mathcal{K}$-bit secret key is also broken down into blocks of 51-bits length accordingly. The number of 3-qubit permutations in the Permutation Pad to achieve 256 bits of entropy is 17, as shown in Table 1.

The reader can consult Fig. 1 and Fig. 5 for the illustration of the general logic of the 3-qubit QPP circuits, however, with updated values $i = 51$, $n = 102$, and most importantly the permutation operators as well dispatching operators will act on 3 qubits at a time. The reader can also use Algorithms 1 and 2 to see the general logic of 3-qubit QPP encryption and decryption circuits respectively, however, with updated values $i = 51$, and $k = 17$.

4 Quantum secure encryption

In this section, we discuss the currently realizable, not ideal, implementation of QPP that we have successfully executed on the IBM quantum computers using the Qiskit development kit. Three major hurdles that make the ideal implementation currently impossible are the lack of quantum channel, the limited capacity of currently publicly available free-of-charge quantum computers, and the noise. We discuss these matters in the next section.

4.1 Noisy quantum computers and quantum volume

It is well understood that although there has been a significant advancement in the field of quantum computing over the years, including demonstrations of quantum supremacy [29, 30], current quantum computers are a far way from the universal fault-tolerant fully-scalable quantum computers. One reason for that is the noise, which causes the current computers to experience errors. Another reason is the capacity of quantum computers in terms of qubits and the size of the largest circuit it can process successfully, in other words without too many errors as to alter the results.

We want to focus the reader's attention on the latter. IBM first introduced the notion of quantum volume (QV) metric to qualify and compare the capabilities of quantum devices [31]. The QV accounts for many factors contributing to the performance of a quantum computer such as the number of qubits, systemic errors, device connectivity, and compiler efficiency [32]. Thomas Lubinski *et al.* generalized the notion of QV to be a function of the job size, in other words, circuit width and circuit depth, that a quantum computer can run without errors.

IBM provides information on quantum volume for each of their quantum computers. The IBM quantum computers that we have used for implementation is `ibmq-manila` and `ibmq-bogota`. Both of these quantum computers have a capacity of 5 qubits and a Quantum Volume of 32. A device's QV is said to be 2^n, where n is the number of qubits or width of the circuit it can execute successfully, and also the number of layers or the depth of the largest circuit it can execute without too much noise as to alter the results or introduce significant errors. Thus, QV of 32 refers to the ability of a quantum computer to successfully run a job corresponding to the largest circuit with 5 layers and 5 qubits [32].

Quantum Volume and qubit capacity are not the only measures of performance that IBMQ makes available to the users. Typically, the providers would include 1- and 2- qubit gate fidelity, state preparation and measurement fidelity, and T1 and T2 coherence times. We include the said information provided by IBMQ for one of the quantum computers we used for implementation, `ibmq-manila`, in Fig. 6. We have tested `ibmq-manila`

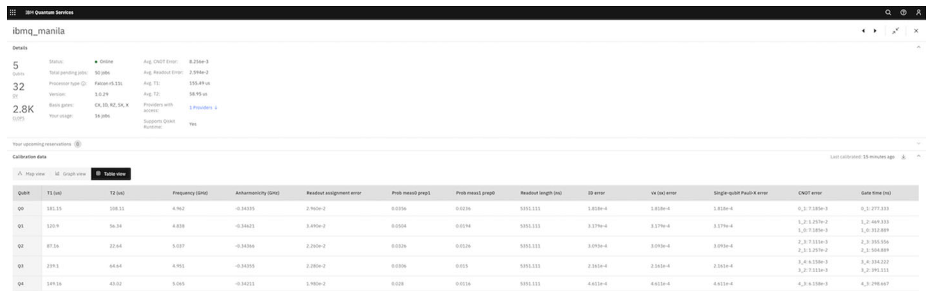

Figure 6 Detailed specifications of the IBMQ "ibmq-manila" quantum computer provided by IBM to its users

for 2-qubit, 3-qubit, and 4-qubit QPP and found the depth of decomposed layers of elementary gates is far beyond its QV for 4-qubit permutation gates. So we will limit our implementations only to 2- and 3-qubit QPP.

Each encryption and decryption circuit that we run on `ibmq-manila` and `ibmq-bogota` for 2-qubit QPP has width of 2, and depth of 3. Interested reader can check this value by editing the source code given in the Appendix. Lines 124–133 can be commented out, and a single command `qc.depth()` can be added. This command will return the depth of a given circuit, which can be appended to the `list_of_ciphers` and printed out. The width of the encryption and decryption circuits of 3-qubit QPP is 3, and the depth is 3 as well. However, corresponding transpired circuits have depth ranging from 5 to 13. Note that increased depth does not significantly affect the results due to the width being only 2 and 3. More detailed discussion on this is available in [32].

4.2 2-qubit QPP circuits

We have successfully implemented 2-qubit QPP on the IBM quantum computers `ibmq-manila` and `ibmq-bogota`, using the Qiskit developmental tool. This implementation uses Permutation Pad consisting of 56 of 2-qubit permutations, however, we encrypt and decrypt only 2 qubits at a time. This is done to account for the limitations of the currently publicly available free-of-charge IBM quantum computers. Indeed, these quantum systems have at most 5 qubits capacity with a Quantum Volume of 32.

To begin communication, parties A and B must agree on a symmetric pre-shared key to be used for the (Inverse) Permutation Pad generation, randomization, as well as the dispatching. For our implementation we will use a classical secret key due to the limited capacity of the quantum computers used for our implementation. Note that, if we were to use qubits to store secret key blocks, each circuit used to encrypt a single plaintext block will consist of 4 qubits. That would result in uninterpretable measurement results, tampered by noise. We, also, use classical randomization and dispatching procedures due to the same limitations.

4.2.1 Permutation pad generation

Assume that both communicating parties have pre-shared a classic secret key $\mathcal{K}$. To achieve 256 bits of entropy, we require that $\mathcal{K}$ is 448 bits long as shown in Table 1. Both communicating parties would divide $\mathcal{K}$ into blocks, to be stored in the list `secret_key_blocks`, of 8 bits each. A simple `for` loop to populate an empty list can do the trick. Each such block is later used to generate one 2-qubit permutation gate. The

Permutation Pad as well as the Inverse Permutation Pad consist of 56 permutation operators each. We allow for repetitions in the (Inverse) Permutation Pad, since the 2-qubit permutations are elements of the symmetric group S_4 of order 4!, so there are at most 4! = 24 possible distinct 2-qubit permutations.

Both communicating parties can use the Fisher-Yates shuffling algorithm to generate the Permutation Pad and the Inverse Permutation Pad using the mentioned secret key blocks as follows. First, the Fisher-Yates algorithm is used to create a shuffled array, given a secret key block. Then, both communicating parties will use a `Numpy np.zeros` routine to create a matrix of zeros. Each such matrix will be edited using the shuffled array created by the Fisher-Yates algorithm and a simple array of n numbers to assign 1 to every row of the matrix in distinct columns. The resulting matrices are permutation matrices. Now, the encrypting party will use command `Permutation_Pad.append(Operator(my_matrix))` to populate the empty list `Permutation_Pad =[]` with permutation matrices converted to quantum operators. The decryption party will use the command `Inverse_Permutation_Pad.append(Operator(my_matrix.transpose())` to populate the `Inverse_Permutation_Pad = []` with the respective inverse permutations converted to quantum operators. Note that inverse permutations are precisely the conjugate transposed permutation matrices. However, since the coefficients in the matrices are all integers it is enough to consider the transposed matrices only. At this point party A and party B have created a Permutation Pad and corresponding Inverse Permutation Pad respectively.

4.2.2 Randomization

For our experiment A sends an encrypted picture of Albert Einstein, illustrated in Fig. 7, to B using the 2-qubit QPP. The natural first step for A to begin the communication is to convert the PNG file of Einstein to a bit-string. That is, using `Image` module from

Figure 7 The picture of Albert Einstein to be encrypted and sent to the decrypting party

the Python PIL fork and the `BytesIO` class from the Python IO Module, A can get the values of the image in bytes and later it to a plaintext bitstring. Next, A randomizes the plaintext message using classical XOR operation with the pre-shared key $\mathcal{K}$, and breaks the randomized plaintext bitstring into block of 2 bits each.

4.2.3 Encryption

After the plaintext has been randomized, A generates a permutation selection array, denoted `perm_selection_blocks`, that will be used for dispatching later. Such array is simply the blocks of the secret key of length 6 bits each, converted to integers and evaluated modulo 56. Each one of these integers point out to the position of a permutation in the Permutation Pad. Next, A creates an empty list, called `list_of_ciphers`, which will be later populated with the ciphertext bits.

The party A begins encrypting every block of the randomized plaintext one by one. For that they use a `for` loop. For each randomized plaintext block, A creates a quantum circuit with 2 qubits and 2 classical bits by defining the `QuantumCircuit()` class, and initializes the qubits to create the state $|b_0 b_1\rangle$, where b_0 is the first bit in the corresponding randomized plaintext block and b_1 is the second one, using the `initialize(Statevector.from_label())` command. A, then, applies the jth permutation from the Permutation Pad using the `append` command, where j = `perm_selection_blocks[x% len(perm_selection_blocks)]` and x is the position of the plaintext block in the array of all the plaintext blocks. This produces a state corresponding to the ciphertext block. Due to inability to send the qubits over a quantum channel, the said block is then measured.

In order to run the job described above in today's noisy quantum computers, each circuit created for each plaintext block needs to be transpiled using the `transpile()` command. Indeed, most input circuits require rewriting to match the topology of a specific quantum device. That is, to make them compatible with a given target quantum computer. A also performs heavy optimization on the circuits. After the circuit is transpiled, A specifies the quantum system which they want to use to execute the job and obtains the result using the `job = execute()` and `result = job.result()` commands. A, then, gets histogram data from the experiment using `counts = result.get_counts()` and stores the highest probability value in the `list_of_ciphers`. Such list contains all the classical ciphertext blocks.

To transmit the ciphertext, A creates a binary file from the data stored in the `list_of_ciphers`. That is, A can join the entries of the `list_of_ciphers` and create ciphertext bytes using the `bytes()` method. A then opens a new binary file and writes the ciphertext bytes to it. A transmits such file over a reliable classical channel.

The illustration of the described encryption procedure is depicted in Fig. 8. We also include the illustration of the ciphertext binary file converted to pixels using an available online tool in Fig. 9. The illustration of the transpiled circuit diagram used for encryption of the first 2-qubits plaintext block is available in Fig. 10 as well as the histogram plot of the measurement result in Fig. 11.

4.2.4 Decryption

Recall, that the decrypting party has already generated an Inverse Permutation Pad corresponding to a Permutation Pad used for the encryption. They, then, perform the

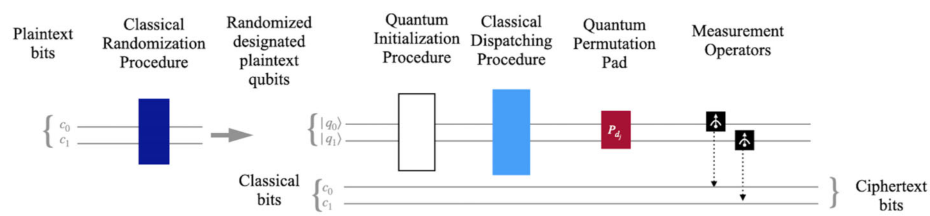

Figure 8 The encryption circuit for functional currently-realizable implementation of a 2-qubits QPP on IBMQ using Qiskit starts by breaking the randomized plaintext bits into 2-bits chunks. Each such chunks is encrypted separately with a new circuit. We give such circuit for one block in this figure. Note that the qubits in each such circuit are initialized according to the randomized plaintext bits of the corresponding bit block

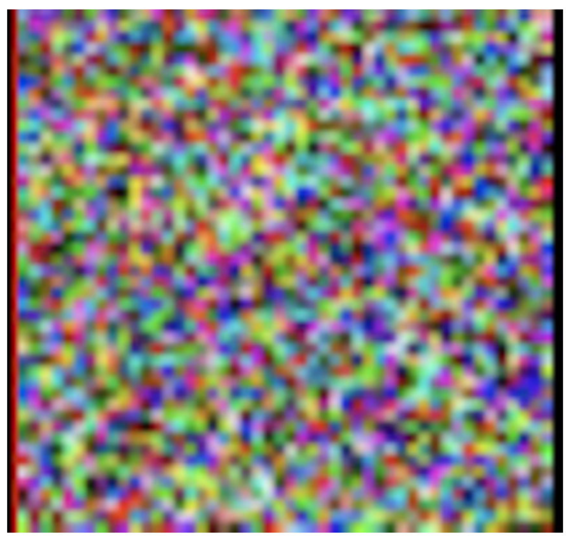

Figure 9 The illustration of the ciphertext binary file converted to raw pixels

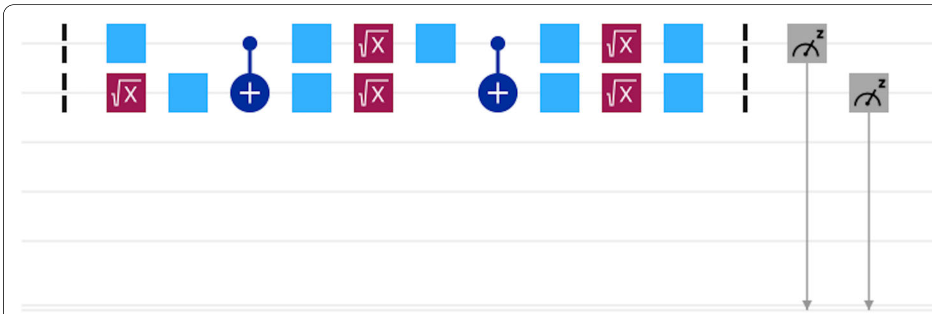

Figure 10 The diagram of a transpiled circuit used to encrypt the first 2-qubits chunks of the plaintext using 2-qubits QPP

exact same procedure as the encrypting party to generate the array `perm_selection_blocks`. Next, the decrypting party B receives the ciphertext binary file, opens it, and separates the ciphertext into blocks of 2 bits each. B creates an empty list called `list_of_messages` to be populated later on.

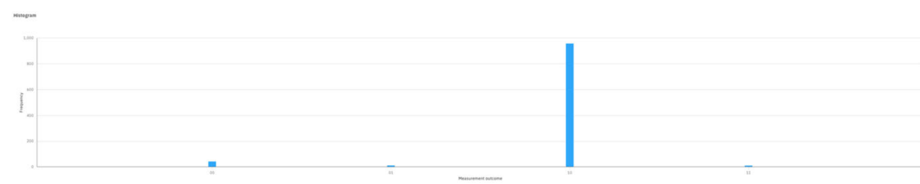

Figure 11 The plot histogram of the measurement results of the encrypted first 2-qubits block of the randomized plaintext using 2-qubits QPP. That is, the measurement results of the first 2-qubits block of ciphertext

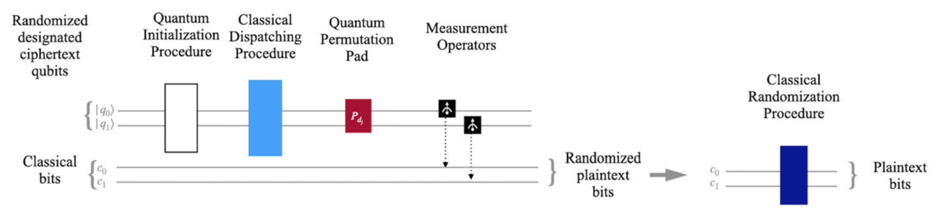

Figure 12 The decryption for functional currently-realizable implementation of a 2-qubits QPP on IBMQ using Qiskit acts on the 2-bit ciphertext blocks. For each such block, a new quantum circuit is created with qubits initialized to the ciphertext block bits. Ciphertext qubits are the decrypted using Inverse QPP and the resulting state is measured. The measurement results are precisely the blocks of the randomized plaintext. The decryption party then applied classical XOR opertaor using the same shared key as the encryption party to produce the original plaintext

Each block of ciphertext is decrypted separately, using the `for` loop. For each block of the ciphertext B creates a quantum circuit with 2 qubits and 2 classical bits by defining the `QuantumCircuit()` class. The qubits are then initialized according to the corresponding ciphertext bits using the `initialize(Statevector.from_label())` command. To the created state B applies jth permutation from the Inverse Permutation Pad using the `append` command, where `j = perm_selection_blocks [x%len(pad_selection_blocks)]`, and the value x is the position of the plaintext block in the array of all the plaintext blocks. That is, this step of the decryption procedure is identical to the same step in the encrypting process but the operator that is acting on the ciphertext block is an inverse permutation taken from the Inverse Permutation Pad. The produced randomized plaintext block is then measured.

Just as the encrypting party A, party B also transpiles each decrypting circuit using the `transpile()` command. B, then, sends the job to be executed on a desired quantum computer and obtains the results. B then gets histogram data from the experiment using the `job.result().get_counts()` command and stores the highest probability value in the `list_of_messages`. Such list contains the binary plaintext. B simply needs to `print("".join(list_of_messages))` to obtain the randomized plaintext binary string.

To de-randomize, B acts with a classical XOR operation with the secret $\mathcal{K}$ on the randomized plaintext.

Lastly, in order to generate a PNG file from the plaintext B opens a new PNG file and writes the binary string to it.

The illustration of the described decrypting procedure is available in Fig. 12. The diagram of the transpiled circuit used for decryption is available in Fig. 13 as well as a plot

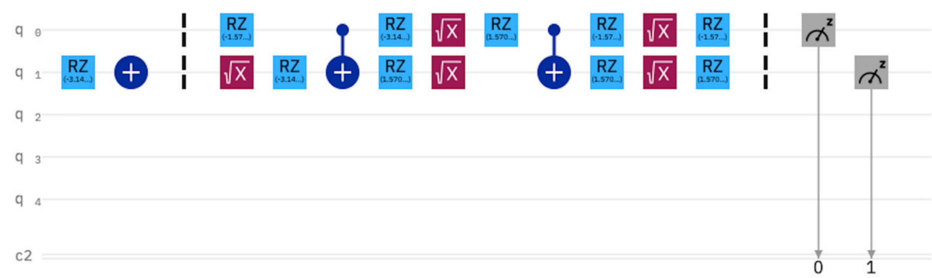

Figure 13 The diagram of a transpiled circuit used to decrypt the first 2-qubits chunks of the ciphertext using 2-qubits QPP

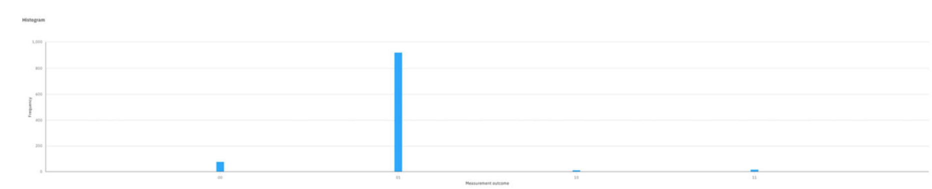

Figure 14 The plot histogram of the measurement results of the decrypted first 2-qubits block of the plaintext using 2-qubits QPP. That is, the measurement results of the first 2-qubits block of randomized plaintrext produced by decryption

histogram of the measurements results in Fig. 14. The source code for the implementation of the 2-qubits QPP can be found in the appendix. The source-code for 3-qubits QPP is very similar except for the parameters as well as number of qubits, permutations in the Permutation Pad and classical bits to store the measurement results. It is available at request to the corresponding author.

4.3 3-qubit QPP circuits

The Quantum Volume and qubits capacity of `ibmq-bogota` and `ibmq-manila` computers allow for a 3-qubit QPP implementation. To provide 256 bits of entropy, this implementation uses a Permutation Pad consisting of 17 3-qubit permutations. Note that due to limitations discussed, the secret key in this implementation is classical as well as the randomization and dispatching procedures. More precisely, this is identical with the 2-qubits QPP implementation except for a few details, which we discuss in the next few sections.

Suppose that party A wants to send the same encrypted picture of Einstein to B, however, now using a 3-qubits QPP.

4.3.1 Permutation pad generation

We list the major differences in the Permutation Pad generation procedure between the two implementations, both providing 256 bits of entropy.

1 The Permutation Pad consists of 17 permutations.
2 The size of the pre-shared secret key is 408 bits.
3 Blocks of the pre-shared key of size 24 bits each are used in the Fisher-Yates algorithm to generate permutation matrices.

4.3.2 Encryption

We list the differences in the encryption procedure between the two implementations.

1 After converting the PNG plaintext file into a binary string, it is divided into blocks of 3 bits each. Note that there are instances of the plaintext that are not divisible by 3 in this case one solution is to add a few bits at the end or beginning of the plaintext until its length is divisible by 3.
2 The dispatching array `perm_selection_blocks` is created using secret key blocks of length 5 bits converted to integers and evaluated modulo 17.
3 The `QuntumCircuit()` created for each plaintext block has 3 qubits and 3 classical bits.

The reader can consult Fig. 8 for the illustartion of the general logic of the 3-qubit QPP implementation circuit, however, with 3 qubits and 3 classical bits instead of 2. Moreover, the operators must be applied to 3-qubit blocks at once instead of 2-qubit blocks.

4.3.3 Decryption

The decryption procedure for the 3-qubit QPP is equivalent to that of 2-qubit QPP except for a few subtle points. These differences are listed in the previous section.

Note that Fig. 12 can be extended to the 3-qubits QPP implementation circuit with 3 qubits and 3 classical bits instead of 2. In addition, the 3-qubits QPP the operators are applied to 3-qubits blocks at once instead of 2-qubits blocks.

4.4 Cipher randomness

The more detailed security analysis of the QPP can be found in [26], however, in this section, we want to present the reader with the ENT test results of the ciphertext produced by encryption on the picture in Fig. 7 using 2- and 3-qubits QPP. We also include the results of the ENT testing of the plaintext itself before the randomization procedure for better comparison.

ENT or Pseudorandom Number Sequence Test Program is a well-recognized and widely used in industry program that tests binary files for the data on information density of the files, or its entropy. In other words, ENT testing determines the randomness of the data in the file. ENT performs a range of tests and output values that indicate the randomness of the data in the file such as Entropy, Arithmetic Mean, the value of Chi-square, Monte Carlo value for π, and the Serial Correlation Coefficient. We provide the data generated by the ENT testing of the ciphertext produced by encrypting Fig. 7 with 2- and 3-qubit QPP as well as the ideal values of the ENT test results in Table 2. We also include the results of the ENT testing of the original pre-randomized plaintext.

Table 2 The table shows data from the ENT testing of the ciphertext produced by encrypting the picture of Einstein using 2- and 3-qubits QPP. We also include ideal values of the ENT testing in the first column for comparison

ENT test values	Ideal Values	2-qubits QPP	3-qubits QPP	AES-256	Plaintext
Entropy (bits)	7.98	7.934235	7.938312	7.936005	7.323068
Chi-square	256	277.57	270.07	250.98	8853.85
p-Value	[0.01, 0.99]	0.1585	0.2471	0.5541	0.0001
Arithmetic Mean	127.50	127.6751	126.1195	129.0124	119.7248
Monte Carlo π	3.141592	3.128107075	3.112810707	2.986899563	3.120458891
Serial Correlation	0	−0.006264	−0.011305	−0.002608	0.244054

For a better comparison, we include ENT test results of the ciphertext produced by AES256 that corresponds to the same plaintext, namely Fig. 7. The secret key for this encryption is a randomly generated string of 256 bits.

The reader can see that the Entropy values of the ciphertext for both QPP implementations are close to the ideal values, and significantly closer than the plaintext value. The entropy value for AES-256 ciphertext is very close to those for QPP ciphertext. There is a notable difference in the values of Chi-square between the QPP encrypted ciphertext and the pre-randomized plaintext. Note that Chi-square value is very sensitive to bias and we can see that the mentioned values for the QPP encrypted ciphertexts are much closer to the ideal value, when the plaintext value is not even in the same ballpark. Chi-square value for AES-256 is closer to the ideal value, however, very similar to the QPP ciphertext values. The p-Value of the plaintext does not fall in the ideal range unlike the QPP encrypted cipheretexts as well as the AES-256 ciphertext. The arithmetic mean values are also much better for the QPP encrypted ciphertexts, with 2-qubits QPP ciphertext being almost identical to the ideal p-Value. As for the Monte Carlo π values, none of the values in columns 2–5 are quite good, however, AES-256 ciphertext is further away from the ideal value than the other values. The Serial Correlation values are much better for the QPP encrypted ciphertexts than the plaintext, and very similar to the AES-256 ciphertext.

Note that it is recommended that ENT testing is done with larger files. In our case, the files are a mere 3KB so we expect the testing results to be better for larger plaintext and ciphertext files.

5 Discussion and conclusion

In this paper, we presented an implementation of Kuang's *et al.'s* Quantum Permutation Pad used to encrypt a PNG file with Qiskit on an IBM quantum computer. Although functioning and providing 256 bits of entropy, this implementation is not an absolute implementation of QPP on a quantum device. We are working on advancing the implementation described in this paper as well as studying other applications of this implementation. For instance, given that QPP can be implemented on a quantum computer as well as classical computer we see its potential for quantum to classical device communication. That is, QPP can be seen as a uniform symmetric encryption algorithm that can be used for a hybrid network with quantum and classical devices. Moreover, if a quantum channel was available, QPP could be shown to be a symmetric encryption scheme that can be used for two quantum computers for communication with no classical parts except for plaintext input bits and plaintext output bits after decryption. Everything in between has a quantum nature.

For our current implementation we used free-of-charge available to public IBMQ computers `ibmq_bogota` and `ibmq_manila` with 5 qubits and a Quantum Volume of 32. To account for the limitations of these computers we implemented QPP using 2- and 3-qubit permutation operators. That is, with permutation operators that act of 2 and 3 qubits at a time respectively. QPP can be implemented using any n-qubits permutations, and as quantum hardware advances we will demonstrate n-qubit QPP. Moreover, for this implementation, we did not use sophisticated randomization and distribution procedures. We use classical operators for randomization and dispatching; In the future, we will be able to use more involved fully-quantum operators for such tasks.

Note that in our previous work [28, 33], we have not included randomization step as well as the dispatching step. A detailed discussion on the importance of these steps is contained in [26]. However, we will point out that although the randomization step already encrypts the plaintext, the same pre-shared key used for randomization can not be reused. However, the Permutation Key Space, as described in [26], can be reused again.

We included the source code for the implementation of 2-qubits QPP in the appendix. The latest source code as well as the source code for 3-qubit QPP is avaialble upon request to the corresponding author. We also conducted ENT tests and described the ENT testing results to show the randomness of the ciphertext after encryption with 2- and 3-qubit QPP.

Appendix

We include a source code for the implementation of the 2-qubits QPP in this appendix. The implementation of 3-qubits QPP is available at request to the corresponding author.

```python
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()

#import any other necessary libraries
from qiskit import execute
from qiskit import QuantumCircuit, assemble, Aer
from qiskit.visualization import plot_histogram, plot_bloch_vector
from qiskit.extensions import RXGate, XGate, CXGate, SwapGate
from qiskit import execute
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import BasicAer
from qiskit.compiler import transpile
from qiskit.quantum_info.operators import Operator, Pauli
from qiskit.quantum_info import process_fidelity
from qiskit.quantum_info import Statevector
from random import seed
from random import randint
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import AerSimulator
from qiskit import IBMQ
from qiskit.compiler import transpile
from qiskit.tools.monitor import job_monitor

# define all the parametres
n = 4     #possible number of quantum states
num_of_bits = 448    #secret key length
bits_in_block = 8    #blocks of 8 bits to generate 56 permutation matrices for
    the pad
num_of_qubits = 2
num_of_perm_in_pad = 56
pad_selection_key_size = 6 #blocks of 6 bits to be used for dispatching

#the secret key is truly random
secret_key = '11100011110001101001000001010111...011011001'
# break the secret key into blocks
secret_key_blocks = [secret_key[i:i+bits_in_block] for i in range(0, len(
    secret_key), bits_in_block)]

# Fisher Yates shuffling
key_chunks = []

def randomize (arr, n):
    for i in range(n-1,0,-1):
        j = key_chunks[i]
        arr[i],arr[j] = arr[j],arr[i]
    return arr

# create a Permutation Pad
Permutation_Pad = []
for num_of_perm in range(num_of_perm_in_pad):
    #bulid an array of elements from 0 to n-1
    my_array = []
    for num in range(n):
```

```python
        my_array.append(num)
    array_of_n_num = my_array.copy()
    #create a shuffled array using Fisher-Yates and secret key blocks
    key_block = secret_key_blocks[num_of_perm]
    key_chunks = [key_block[i:i+num_of_qubits] for i in range(0, len(key_block), num_of_qubits)]
    for num in range(len(key_chunks)):
        key_chunks[num] = int(key_chunks[num],2)
    len_of_array = len(my_array)
    shuffled_array = randomize(my_array, len_of_array)
    #create a matrix of zeros
    matrix_of_zeros = np.zeros((n, n), dtype=int)
    my_matrix = matrix_of_zeros
    #insert '1' at every column but different rows
    for num in range(n):
        my_matrix[array_of_n_num[num]][shuffled_array[num]] = 1
    #populate the Permutation Pad
    Permutation_Pad.append(Operator(my_matrix))

# create an array used for dispatching
pad_selection_blocks = [secret_key[i:i+pad_selection_key_size] for i in range(0, len(secret_key), pad_selection_key_size)]
for num in range(len(pad_selection_blocks)):
    pad_selection_blocks[num] = (int(pad_selection_blocks[num],2)% num_of_perm_in_pad)

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

#define secret key blocks to be used for randomization
key_for_xor_list = []
for x in range(len(message)):
    key_for_xor_list.append(secret_key[x%len(secret_key)])
key_for_xor = "".join(key_for_xor_list)

#randomize the message using XOR opertaion
randomized_message=[str(int(message[i])^int(key_for_xor[i])) for i in range(len(message))]
randomized_message=''.join(randomized_message)

chunk_size = num_of_qubits
#breaking the randomized message into blocks of 2 bits
message_chunks = [randomized_message[i:i+chunk_size] for i in range(0, len(randomized_message), chunk_size)]


list_of_ciphers = []
#start the encryption proccess for each randomized message chunk
for x in range(len(message_chunks)):
    state_vector = message_chunks[x]
     #establish a circuit with 2 qubits and 2 classical bits to store measureemnt results later
    qc = QuantumCircuit(num_of_qubits, num_of_qubits)
    #initialize the qubits according to the randomized message bits
    qc.initialize(Statevector.from_label(state_vector))
    qc.barrier()
    #apply permutation from the pad based on its position in the pad determined by the dispatching array
    j = pad_selection_blocks[x%len(pad_selection_blocks)]
     #apply permutation
    qc.append(Permutation_Pad[j], range(num_of_qubits))
    qc.barrier()
    #measure the result
    qc.measure([0,1], [0,1])
    #transpile the circuit
    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","rccx","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())

#create one unified ciphertext string and convert to bytes
s = "".join(list_of_ciphers)
bytes_cipher = bytes(int(s[i : i + 8], 2) for i in range(0, len(s), 8))
#write the bytes into a binary file
file_cipher = open("ciphertext_to_send.bin","wb")
file_cipher.write(bytes_cipher)
file_cipher.close()

#######################################################
#ciphertext is transmitted
#######################################################

#Inverse Permutation Pad is created
Inverse_Permutation_Pad = []
for num_of_perm in range(num_of_perm_in_pad):
    my_array = []
    for num in range(n):
        my_array.append(num)
    array_of_n_num = my_array.copy()
    key_block = secret_key_blocks[num_of_perm]
    key_chunks = [key_block[i:i+num_of_qubits] for i in range(0, len(key_block), num_of_qubits)]
    for num in range(len(key_chunks)):
        key_chunks[num] = int(key_chunks[num],2)
    len_of_array = len(my_array)
    shuffled_array = randomize(my_array, len_of_array)
    matrix_of_zeros = np.zeros((n, n), dtype=int)
    my_matrix = matrix_of_zeros
    for num in range(n):
        my_matrix[array_of_n_num[num]][shuffled_array[num]] = 1
    #note that the matrix transpose is appended
    Inverse_Permutation_Pad.append(Operator(my_matrix.transpose()))

# ciphertext binary file is read and the content is extracted to be transformed into a binary string
pip install bitstring
from bitstring import BitStream, BitArray
file = open("ciphertext_to_send.bin","rb")
a = BitArray(file.read())
ciphertext = a.bin

#ciphertetx is broken into blocks
chunk_size = num_of_qubits
cipher_chunks = [ciphertext[i:i+chunk_size] for i in range(0, len(ciphertext), chunk_size)]

list_of_messages = []
# the ciphertetx block are decrypted one by one
for x in range(len(cipher_chunks)):
    qc_decrypt = QuantumCircuit(num_of_qubits, num_of_qubits)
    qc_decrypt.initialize(Statevector.from_label(cipher_chunks[x]))
    qc_decrypt.barrier()
    j = pad_selection_blocks[x%len(pad_selection_blocks)]
    # note that the inverse permutations
    qc_decrypt.append(Inverse_Permutation_Pad[j], range(num_of_qubits))
    #from the Inverse Permutation Pad are acting on ci[hertext
    qc_decrypt.barrier()
    qc_decrypt.measure([0,1], [0,1])
    #transpile the circuit
    qc = 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 ciphertext chunks
    provider = IBMQ.load_account()
    qcomp = provider.get_backend('ibmq_bogota')
    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())

 #de-randomize the decrypted ciphertext
randomized_decrypted_cipher = "".join(list_of_messages)
```

```
decrypted_message=[str(int(randomized_decrypted_cipher[i])^int(key_for_xor[i]))
     for i in range(len(randomized_decrypted_cipher))]
decrypted_message=''.join(decrypted_message)

#convert the data into bytes
decrypted_bytes = [int(decrypted_message[i:i + 8], 2) for i in range(0, len(
     decrypted_message), 8)]
#create a PNG file from the decrypted plaintext
with open('decrypted_pic.png', 'wb') as f:
    f.write(bytes(decrypted_bytes))
#Voila!
```

Acknowledgements
Authors acknowledge IBM for their free of charge 5-qubit quantum computers used for this study.

Abbreviations
QPP, Quantum Permutation Pad; QKD, Quantum KeyDistribution; DV-QKD, Discrete Variable QKD; CV-QKD, Continuous Variable QKD; TF-QKD, Twin-field QKD; OTP, One-Time-Pad; QV, Quantum Volume; CNOT, Control-NOT gate; PRNG, Pseudo Random Number Generator; QRNG, Quantum Random Number Generator; pQRNG, Pseudo QRNG; QSDC, Quantum Secure Direct Communication.

Availability of data and materials
Partial data generated or analysed during this study are included in this published article and its supplementary information files. Any datasets used and/or analysed during the current study that have not been included in this published article or its supplementary information files are available from the corresponding author on reasonable request.

Declarations

Competing interests
The authors declare no competing interests.

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 and Mrs. Perepechaenko contributed on the implementation of QPP in IBMQ systems described in this paper.

Authors' information
Quantropi Inc., Ottawa, ON, Canada, ON K1Z 8P9.

Publisher's Note

Received: 5 April 2022 Accepted: 29 September 2022 Published online: 12 October 2022

References
1. Bennett CH, Brassard G. Quantum cryptography: public key distribution and coin tossing. Theor Comput Sci. 2014;560:7–11. https://doi.org/10.1016/j.tcs.2014.05.025.
2. Djordjevic IB. Discrete variable (DV) QKD. In: Physical-layer security and quantum key distribution. Berlin: Springer; 2019. https://doi.org/10.1007/978-3-030-27565-5_7.
3. Lai J-S, Lin X-Y, Qian Y, Liu L, Zhao W-Y, Zhang H-Y. Deployment-oriented integration of dv-qkd and 100g optical transmission system. In: Asia communications and photonics conference (ACP). vol. 2019. Chengdu: IEEE; 2019. p. 1–3. http://opg.optica.org/abstract.cfm?URI=ACPC-2019-T2H.1.
4. Qi B. Bennett-brassard 1984 quantum key distribution using conjugate homodyne detection. Phys Rev A. 2021;103:012606. https://doi.org/10.1103/PhysRevA.103.012606.
5. Pirandola S, Mancini S, Lloyd S, Braunstein SL. Continuous-variable quantum cryptography using two-way quantum communication. Nat Phys. 2008;4(9):726–30. https://doi.org/10.1038/nphys1018.
6. Pirandola S, García-Patrón R, Braunstein SL, Lloyd S. Direct and reverse secret-key capacities of a quantum channel. Phys Rev Lett. 2009;102(5):050503. https://doi.org/10.1103/physrevlett.102.050503.
7. Weedbrook C, Pirandola S, García-Patrón R, Cerf NJ, Ralph TC, Shapiro JH, Lloyd S. Gaussian quantum information. Rev Mod Phys. 2012;84(2):621–69. https://doi.org/10.1103/revmodphys.84.621.
8. Lucamarini M, Yuan ZL, Dynes JF, Shields AJ. Overcoming the rate–distance limit of quantum key distribution without quantum repeaters. Nature. 2018;557(7705):400–3. https://doi.org/10.1038/s41586-018-0066-6.
9. Chen J-P, Zhang C, Liu Y, Jiang C, Zhang W-J, Han Z-Y, Ma S-Z, Hu X-L, Li Y-H, Liu H, Zhou F, Jiang H-F, Chen T-Y, Li H, You L-X, Wang Z, Wang X-B, Zhang Q, Pan J-W. Twin-field quantum key distribution over a 511 km optical fibre linking two distant metropolitan areas. Nat Photonics. 2021;15(8):570–5. https://doi.org/10.1038/s41566-021-00828-5.
10. Wang S, Yin ZQ, He DY et al. Twin-field quantum key distribution over 830-km fibre. Nat Photonics. 2022;16:154–61. https://doi.org/10.1038/s41566-021-00928-2.
11. Deng F-G, Long GL, Liu X-S. Two-step quantum direct communication protocol using the Einstein-Podolsky-Rosen pair block. Phys Rev A. 2003;68(4):042317. https://doi.org/10.1103/physreva.68.042317.

12. Deng F-G, Long GL. Secure direct communication with a quantum one-time pad. Phys Rev A. 2004;69(5):052319. https://doi.org/10.1103/physreva.69.052319.
13. Zhang W, Ding D-S, Sheng Y-B, Zhou L, Shi B-S, Guo G-C. Quantum secure direct communication with quantum memory. Phys Rev Lett. 2017;118(22):220501. https://doi.org/10.1103/physrevlett.118.220501.
14. Langenberg B, Pham H, Steinwandt R. Reducing the cost of implementing the advanced encryption standard as a quantum circuit. IEEE Trans Quantum Eng. 2020;1:1–12. https://doi.org/10.1109/TQE.2020.2965697.
15. Wang Z, Wei S, Long G. A quantum circuit design of AES. 2021. arXiv:2109.12354.
16. Zou J, Wei Z, Sun S, Liu X, Wu W. Quantum circuit implementations of aes with fewer qubits. 2020. https://doi.org/10.1007/978-3-030-64834-3_24.
17. Jang K, Song G, Kim H, Kwon H, Kim H, Seo H. Efficient implementation of present and gift on quantum computers. Appl Sci. 2021;11(11):4776. https://doi.org/10.3390/app11114776.
18. Baksi A, Jang K, Song G, Seo H, Xiang Z. Quantum Implementation and Resource Estimates for RECTANGLE and KNOT. Cryptology ePrint Archive, Report 2021/982. 2021. https://ia.cr/2021/982.
19. Hu Z, Kais S. A quantum encryption design featuring confusion, diffusion, and mode of operation. Sci Rep. 2021. https://doi.org/10.1038/s41598-021-03241-8.
20. Kuang R. Methods and systems for data protection. Google Patents. 2019. US Patent 10476664. https://patentimages.storage.googleapis.com/07/0a/5b/82e9fd00a38e08/US10476664.pdf.
21. Kuang R. Methods and systems for secure data communication. Google Patents. 2022. US Patent 11323247. https://patentimages.storage.googleapis.com/13/68/bb/b21a2b559881c3/US11323247.pdf.
22. Kuang R, Bettenburg N. Shannon perfect secrecy in a discrete Hilbert space. In: 2020 IEEE international conference on quantum computing and engineering (QCE). 2020. p. 249–55. https://doi.org/10.1109/QCE49297.2020.00039.
23. Kuang R, Lou D, He A, Conlon A. Quantum safe lightweight cryptography with quantum permutation pad. In: 2021 IEEE 6th international conference on computer and communication systems (ICCCS). 2021. p. 790–5. https://doi.org/10.1109/ICCCS52626.2021.9449247.
24. Kuang R, Lou D, He A, Conlon A. Quantum secure lightweight cryptography with quantum permutation pad. Adv Sci Tech Eng Syst J. 2021;6(4):790–5. https://doi.org/10.25046/aj060445.
25. Lou D, Kuang R, He A. Entropy transformation and expansion with quantum permutation pad for 5g secure networks. In: 2021 IEEE 21st international conference on communication technology (ICCT). 2021. p. 840–5. https://doi.org/10.1109/ICCT52962.2021.9657891.
26. Kuang R, Barbeau M. Quantum permutation pad for universal quantum-safe cryptography. Quantum Inf Process. 2022;21:211. https://doi.org/10.1007/s11128-022-03557-y.
27. Kuang R, Lou D, He A, McKenzie C, Redding M. Pseudo quantum random number generator with quantum permutation pad. In: 2021 IEEE international conference on quantum computing and engineering (QCE). 2021. p. 359–64. https://doi.org/10.1109/QCE52317.2021.00053.
28. Perepechaenko M, Kuang R. Quantum encrypted communication between two ibmq systems using quantum permutation pad. In: 2022 11th international conference on communications, circuits and systems (ICCCAS). 2022. p. 146–52. https://doi.org/10.1109/ICCCAS55266.2022.9824836.
29. Zhong H-S, Wang H, Deng Y-H, Chen M-C, Peng L-C, Luo Y-H, Qin J, Wu D, Ding X, Hu Y, Hu P, Yang X-Y, Zhang W-J, Li H, Li Y, Jiang X, Gan L, Yang G, You L, Wang Z, Li L, Liu N-L, Lu C-Y, Pan J-W. Quantum computational advantage using photons. Science. 2020;370(6523):1460–3. https://doi.org/10.1126/science.abe8770.
30. Arute F, Arya K, Babbush R, Bacon D, Bardin J, Barends R, Biswas R, Boixo S, Brandao F, Buell D, Burkett B, Chen Y, Chen Z, Chiaro B, Collins R, Courtney W, Dunsworth A, Farhi E, Foxen B, Martinis J. Quantum supremacy using a programmable superconducting processor. Nature. 2019;574:505–10. https://doi.org/10.1038/s41586-019-1666-5.
31. Cross AW, Bishop LS, Sheldon S, Nation PD, Gambetta JM. Validating quantum computers using randomized model circuits. Phys Rev A. 2019;100:032328. https://doi.org/10.1103/PhysRevA.100.032328.
32. Lubinski T, Johri S, Varosy P, Coleman J, Zhao L, Necaise J, Baldwin CH, Mayer K, Proctor T. Application-oriented performance benchmarks for quantum computing. 2021. https://doi.org/10.48550/ARXIV.2110.03137. https://arxiv.org/abs/2110.03137
33. Perepechaenko M, Kuang R. Quantum encryption and decryption in IBMQ systems using quantum Permutation Pad. J Commun. 2022. Unpublished.