Quantum Key Distribution

Description:

Quantum key distribution (QKD) aims to distribute a key to two parties across an unsecured channel. We can use this protocol to detect if an eavesdropper has tried to read communications between the parties, with a good probability if long messages are sent.

[A step-by-step guide can be found in the Qiskit tutorials with a more detailed description:](https://qiskit.org/textbook/ch-algorithms/quantum-key-distribution.html)

1. Alice generates a random array of bits.
2. Alice generates a random array of basis.
3. Alice encodes the bits using the generated basis into qubits.
4. Alice sends the encoded qubits to bob across the unsecured channel.
5. Bob generates a random array of basis.
6. Bob decodes (measures) the qubits using the random basis he generated.
7. Alice and Bob share the basis they used.
8. Where the bits in pair of basis matches, we keep the (decoded for Bob or original for Alice) value of the key.
9. We share a sample of this key across the unsecured channel, if they match and they are of a good length, it is unlikely that anyone has copied the key.

Why this works:

If Eve – an eavesdropper – tries to copy or measure the encoded qubits, she will have to also use a randomly generated array of basis.

A fundamental rule of quantum computing is that we cannot copy a qubit that is in an arbitrary quantum state. In other words, to be able to copy a qubit, we need to have prior knowledge on the state of the qubit, which Eve does not have.

Eve also cannot measure the qubits and send them over to Bob, as the qubits are encoded in different basis, and if we try to measure a qubit using the wrong basis, we will collapse the superposition. Changing the state of the qubits that arrive to Bob.

Side-by-side implementations of QKD

|  |  |  |
| --- | --- | --- |
| Qiskit | Cirq | Q# |
|  |  |  |
| Used in multiple steps, generate a binary message. | 🡨 | 🡨  (Annoying syntax but this seems like the easiest way in Q#) |
|  |  |  |
| Define a function that creates an array of circuits, each circuit contains one of the bits of the message to send. | 🡨 | 🡨  We have to pass in the qubits that we intend to use for encoding. |
|  |  |  |
| In this step we encode the message in an array of 1 qubit circuits.  We use the ‘bases’ binary array to determine the base we encode the message. (If the value is set to ‘1’ in the bases array, we apply the h gate)  We use the ‘message’ binary array to determine the position of the qubit, if its in the |0> or |1> state before we apply the h gate | 🡨  As an extra note: we use the Identity gate (I) instead of ‘pass’ for the decoding step later. If we do not pass in the qubits through a gate in the circuit, we cannot later retrieve them through the circuit, and would not be able to measure the circuit. | 🡨  It is more challenging to create an array of length 1 circuits in q#, so we define the circuits at the same time. But the same logic applies |
|  |  |  |
| Method to measure the encoded message. | 🡨 | 🡨 instead of passing the array of circuits with ‘message’, we pass in the array of qubits |
|  |  |  |
| Place the qubits in the circuit into the base according to the basis array by applying the h gate.  Then apply the measurement gate | 🡨  We also call the all\_qubits() method to get the qubits, so we can apply the gates.  (we do not need to do this in qiskit, as we only need the index of the qubit, but we need the actual qubit object in cirq) | 🡨 |
|  |  |  |
| Set up the backend to measure the circuit, format and return them. | 🡨 | Need to reset all qubits after we perform the M operation on them, otherwise Q# will throw an exception |
|  |  |  |
| Function that compares the bases and bits, in order to get an array containing only the matching bits. | 🡨 | 🡨 |
|  |  |  |
| Return a sample of the matching bits, in order to compare the samples and check if the communication was intercepted. | 🡨 | 🡨  Implementation is again, awkward due to the Q# syntax |
|  |  |  |
| Simple driver code:  Check the link with the steps, we are just calling the functions in order and comparing the outputs. | 🡨 | 🡨  Only difference is that we create the qubit array here instead of inside the encodeMessage() method |
|  |  |  |
|  |  | Need to define this method as I could not find a prebuilt method that did the same thing. |