BB84 Protocol

Link to the original code and explanation:

<https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/Cryptography/QuantumKeyDistribution.ipynb>

## About, in a nutshell

In cryptography, key exchange between involved parties must happen without losing confidentiality, availability and integrity. Often, it is necessary to use a public channel to exchange keys, which may be exposed to many risks.

The most used algorithm for key distribution is RSA, but now that we have quantum computers and new algorithms, new security measures are needed.

Therefore, it's crucial to find new alternatives for key distribution, like the BB84 protocol.

BB84 is a protocol for key distribution now that we have quantum computers.

It is an alternative to RSA and other algorithms and protocols used in classical computer science for similar purposes.

An important advantage of using BB844 protocol is that **if an attacker intercepts the key during the exchange, they cannot read it**, as it would mean they have to measure it, therefore making the quantum system collapse into a classical state, with a consequent loss of information.

The requirements to use this protocol are:

* All parties must have access to their own quantum computer.
* They must have a communication channel capable of transmitting qubits.
* They must have a classical communication channel.

Since it is impossible to ensure perfect security, we must assume that any of these channels can be tapped into by an attacker.

If Alice and Bob want to exchange a key, in a nutshell:

* Alice creates a random string of bits, and for each bit, she randomly chooses a basis to encode it in.
* Alice encodes the bits into qubits using her chosen bases, and sends the qubits over a quantum communication channel to Bob's quantum computer.
* Bob also randomly chooses a basis to decode each qubit in. He measures each qubit in the bases he chose.
* Alice uses a classical communication channel to tell Bob which bases she chose. She also tells him the first few bits she sent.
* Bob analyzes these first few bits to determine whether Eve tapped into their quantum communication channel and intercepted Alice's qubits.
* If Eve did not intercept the qubits, they consider all of the qubits that they happened to choose the same bases for, and use those bits as their key. If Eve did intercept the qubits, they repeat the process all over again.

## Instructions on how to run the program

There are two ways to execute quantum\_key\_distribution.ipynb:

* It is possible to run quantum\_key\_distribution.ipynb more than once, even lots of times, using execution\_in\_loop.py and giving it the right inputs only at the beginning, as they will be applied to each execution of quantum\_key\_distribution.ipynb. It may take some time to obtain the results, though.
* Otherwise, quantum\_key\_distribution.ipynb can be run by itself and only one execution of the algorithm will be done.

Examples of inputs for execution\_in\_loop.py and more details about how it works are in the comments in the file.

In quantum\_key\_distribution.ipynb there are two parts of the code and, therefore, two executions of the BB84 protocol: the possibility of an attack is considered only in the second part, as it was in the original code from GitHub.

## Analyzing obtained data

For every single execution of quantum\_key\_distribution.ipynb, two cases will be analyzed:

* When everything should be fine
* When an eavesdropper tries to intercept the key

Results of elaborations are saved in results\_of\_executions.csv by default (but they can also be saved on another file) in order to be able to confront the relevant aspects of the obtained results.

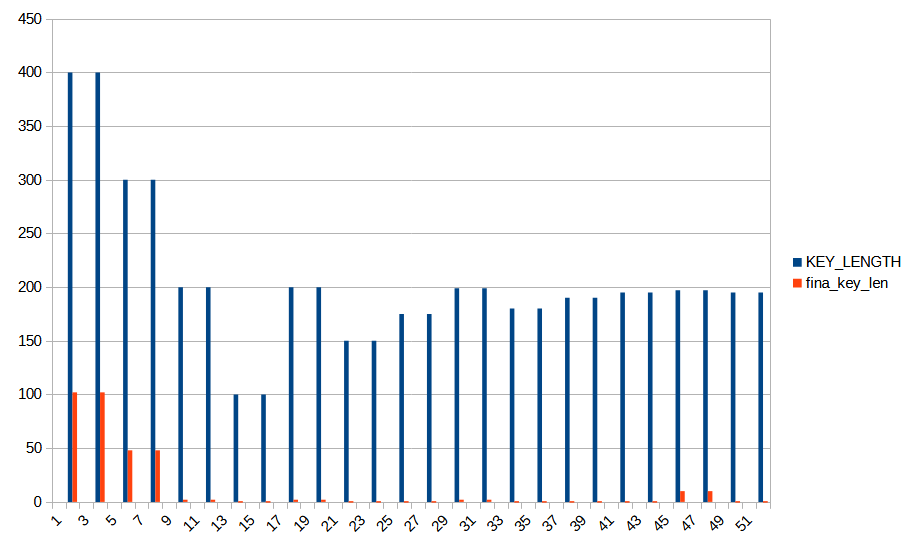
On every csv file generated:

* KEY\_LENGTH: the number of bits at the beginning
* job\_id: it can either contain ‘simulator’ or the ID of the job
* final\_key\_len: the number of bits in the end (the effective length of the generated key)
* attack\_noticed: whether in that execution was considered or not the possibility of an attack
  + possible values in ‘attack\_noticed’: not considered, true or false
* timestamp: recording timestamps can help with noticing how long the program took to be executed, given a number of times.

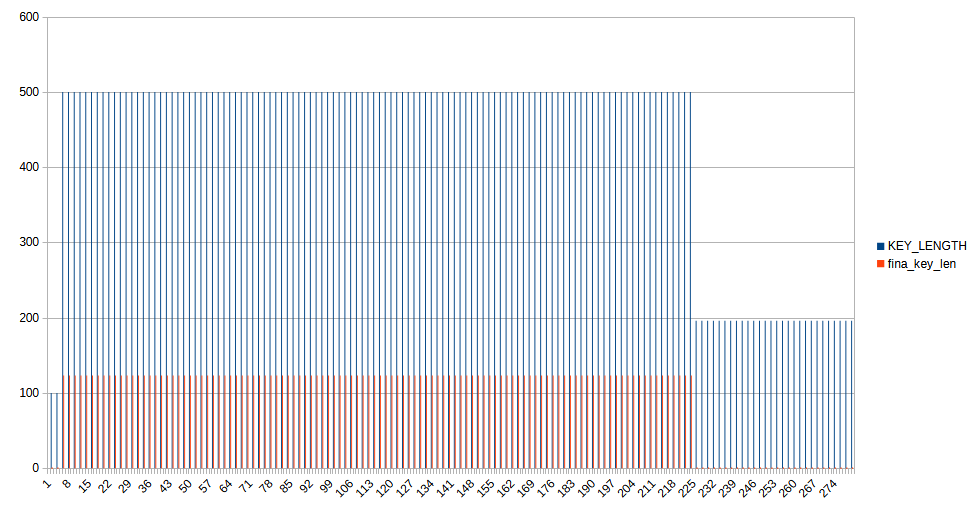
## Results

In order to optimize the use of the real quantum computer, I ran various examples on the simulator, confronting various parameters such as how the length of the key changes given an initial length or checking how long the executions took.

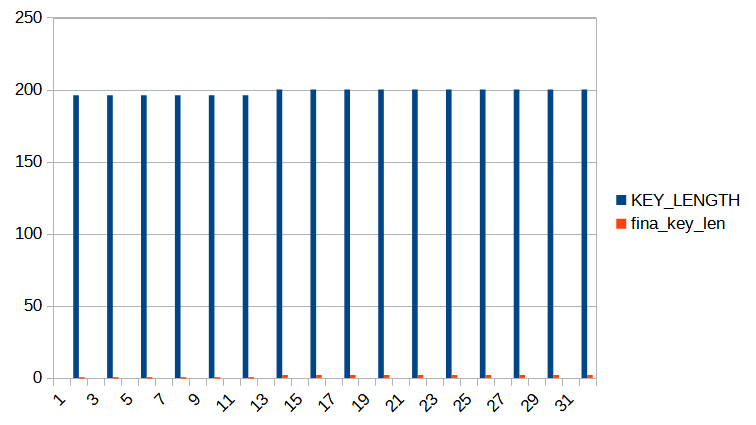
### Random seed: 0 (as in the example)



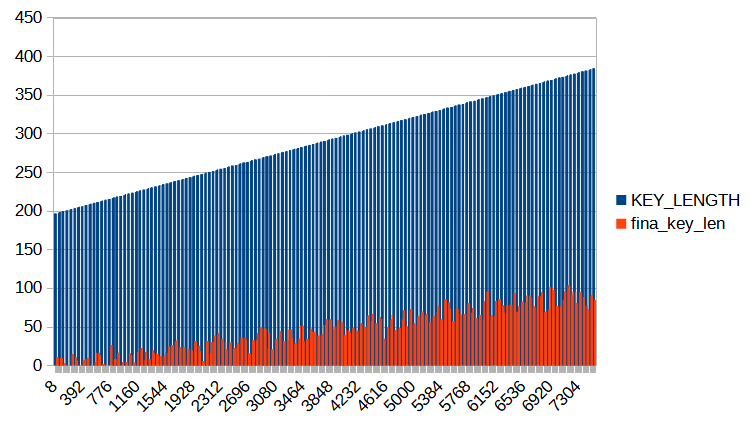
Confronting initial and final key length (random seed used: 0, as in the given example)



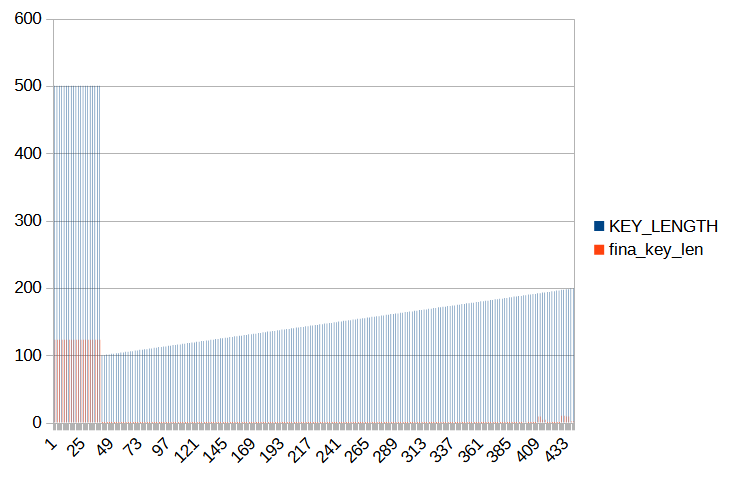
In particular: In executions with 196 bits at the beginning, key went from 196 to 0. Most of the values below 196 at the beginning seem to give this output. (random.seed: 0 again)



Other confrontations between key lengths before and after the executions. (random.seed: 0 again)



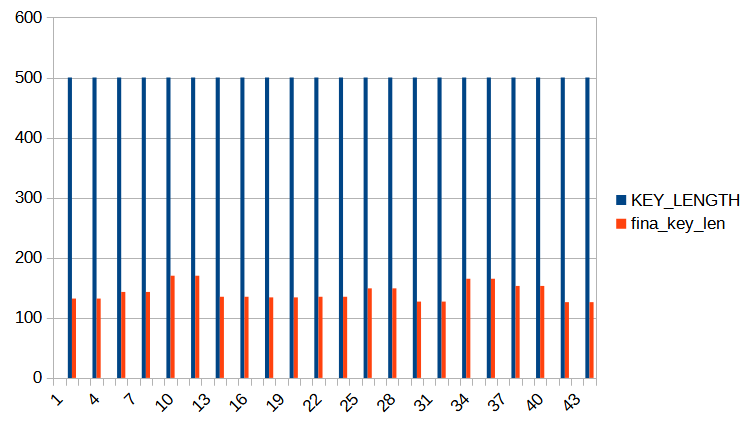
Executions with initial key length from 196 to 384. For each key length, the algorithm has been executed 10 times (random.seed: 0 again)

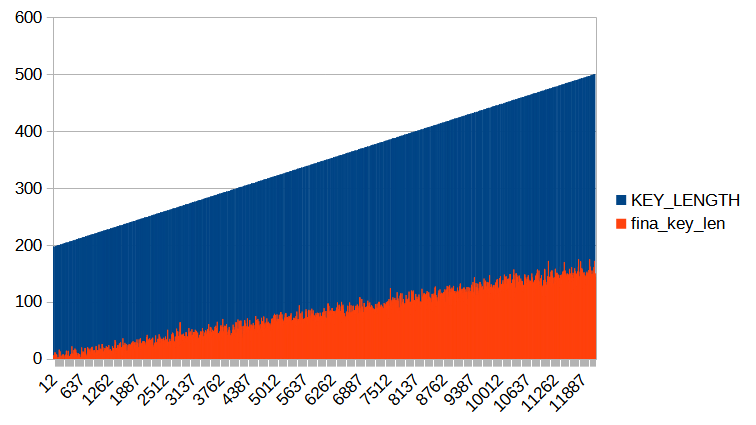


Key length = 500 vs Key length in range 100-199 (random.seed: 0). As you can see, it is necessary to start with a long series of bits in order to obtain a safe enough key.

### Random seed: not initialized

After changing the program in order to use a random seed and generate more real-like data:





(From 197 to 500 initial key length)

### Execution on the real quantum computer

With the free plan, I could only execute on the real quantum computer once as every job took 3s (everytime I executed the measure function). In order to complete the program execution at least once, I executed it once with the lowest initial key length.

KEY\_LENGTH,job\_id,fina\_key\_len,attack\_noticed,timestamp

197,ct5bg99kmkz00086n130,2,not to consider,2024-07-07 17:46:00

## Conclusions

It’s possible to draw the following conclusions observing the obtained results:

* with initial key length being 196 or lower, the final key length obtained was generally 0. Also, trying to use values that are too low may lead to problems in the generation of the key itself;
* It is necessary to start with a long series of bits in order to obtain a safe enough key;
* In every case where the possibility of an attack was considered, every attack (eavesdropping) made by Eve was detected by Alice and Bob, who had the chance to change the key.

### Other possible vulnerabilities

While the confidentiality of the key can be guaranteed on the quantum communication channel, there are other possible vulnerabilities to consider when analyzing the BB84 protocol:

* Availability and integrity of the transmitted data:
  + Eve can continue intercepting the data on the quantum communication channel, making it impossible to generate a new key due to continuous measurements (Denial Of Service)
  + Eve can manipulate the basis related information on the classical channel, leading to an incorrect key generation.
  + Eve can compromise (for example with malware) the involved devices and therefore can also compromise Alice’s or Bob’s bits and/or bases so that they always know what the communication channels contain.
    - An example could be forcing a certain value for random.seed() before the execution with a replay attack, if there’s a vulnerability in the implementation that permits it.
  + Both classical devices and communication channels can be compromised by Eve
* Authentication related vulnerabilities:
  + Eve can imperson Alice or Bob and communicate with the other one. This way, the first may see their service denied (DoS) and the other may not notice the intrusion, exchanging information with Eve.
  + Also, Eve could be a Man-in-the-middle and position themselves between Alice and Bob, substituting their communication. This allows Eve to intercept and manipulate messages, without Alice and Bob noticing because Eve can make it seem that no one is interfering.