COMPX304 Assignment 2: Quantum Key Exchange

Stefenie Pickston

28-04-2022

Abstract

In order to encrypt information over an insecure channel, a symmetrical encryption scheme such as a XOR cipher can be used. However, in a symmetrical encryption scheme both the sending and recieving parties must have the same key, which must also be transmitted over the insecure channel without being intercepted by a malicious eavesdropper.

A method to solve this problem is to use a quantum key exchange, where qubits are exchanged instead of regular bits. Qubits are subject to quantum physics where they have two different types of polarisations, and thus must be transferred over a quantum communications chanel.

The assignment uses a 'simplified' version of qubits to mimic quantum mechanics.

This report will cover the software design of a quantum key exchange algorithm, a man in the middle attack on it, the tests conducted and discussion of the test results.

1 Quantum Key Exchange Algorithm

1.1 Software Design

I implemented my solution in Python. I created classes for the client, server, Qubit and XOR encryption, as well as a test script for the initial QKE algorithm and unit tests. The client and server utilises User Datagram Protocol and encodes data using UTF-8.

The client includes methods to send, recieve, encrypt and decrypt messages, as well as methods to send and recieve polarisations. It also implements a *sendqubits()* method where random qubits are generated and then sent to the server. Because I don't have a quantum communication channel, I have to encode the qubits with UTF-8. This means I can't actually pass the qubit objects, so as an alternative I sent two strings of bits, one for the polarisations and the other for the values of the qubits.

The server is similar to the client, however instead of generating qubits, it has a *recievequbit()* method where it takes the qubit polarisations and values generated by the client. It then creates qubits and measures them to generate it's own polarisation and values. These values are also stored in a string of bits.

The Qubit class was created based on the specifications in the assignment PDF. However I did add an extra *getPolar()* method to be able to access the polarisation values to make debugging, unit testing and the man in the middle attack easier.

The XOR class has two methods. The *cipher(key, msg)* method takes two numeric values and performs an XOR operation on them. The *repeatKey(key, length)* method repeats an XOR key so it's bit length matches the bit length of the message input.

1.2 Testing

To unit test the XOR class, I generated a binary key and message. Then I called the *repeatkey(key, length)* method to make the key the same length as the message. Afterwards I called the *cipher(key, msg)* method once on the original message and key, and additionally once on the encrypted message and key. If the outcome of the decrypted message matches the original then the test passes.

To unit test the Qubit class, I created a Qubit with the value of 1 and the polarity of 0. Then I kept measuring the qubit with the opposite polarity. The test would pass once the value returned had flipped to a 0.

To unit test the QKE algorithm, I setup the server and client and transferred a specified stream length (16, 256 and 1024) of Qubits from the client to the server. Both the client and server create their keys and perform an exchange of an encrypted message. If the recieved message matches the message sent then the test passes.

1.3 Evaluation

I used User Datagram Protocol to transfer data as it is more representative of a real life application. However, I did have to take some shortcuts as I did not have access to a quantum communications channel. This included sending the qubits as a string and creating methods to access/edit certain properties of the server and client. Python is also not ideal for bit manipulation so I had to make do with converting data between strings and integers. At the end of the day the resulting programs were enough to demonstrate the process in action.

2 Man in the Middle Attack

2.1 Implementation

To demonstrate a man in the middle attack, I implemented a *MiddleMan* class which acted as a pipe where communications would pass through it before reaching it's destination. I also created a *main.py* file to run and test the exchange.

2.2 Evaluation

The way that my middle man attack was implemented was per the assignment specifications. This meant that the client would send a stream of qubits to the server, which gets intercepted by the man in the middle. The man in the middle then measures the qubits before passing them on to the client. Here the values and polarisation of the qubits may or may not have changed. Then when the client and server exchange polarisations the man in the middle keeps a copy of them.

When running the attack, the QKE algorithm was only somewhat sucsessful against defending the man in the middle. In most cases, the message would become corrupted or garbled for both the man in the middle and the server due to all the keys being different (note I have omitted most of the corrupted output in the examples):

```
Server Key: 100000000000
Middle Man Key: 101000000
Client Key: 100000100
Client Sent Message: Hello There!
Cannot decode UTF-8
MiddleMan Recieved Message: b'\x0c\xed|M-\xa4\\yG6\xed!...'
Cannot decode UTF-8
Server Recieved Message: b'Ld\xfeH/0t!gvd#\...'
```

Note that the corrupted output results from the UTF-8 encoding being unable to decode the input.

Sometimes the man in the middle attack was sucsessful where they received the client's message, however the server would recieve either corrupted output:

```
Server Key: 000111001
Middle Man Key: 01101
Client Key: 01101
Client Sent Message: Hello There!
MiddleMan Recieved Message: Hello There!
Cannot decode UTF-8
Server Recieved Message: b'\xdc\xa2^\xfe\x95\xae\xa60#
\xba\x14\...'
```

Or a garbled message:

```
Server Key: 01100001
Middle Man Key: 01110000
Client Key: 01110000
Client Sent Message: Hello There!
MiddleMan Recieved Message: Hello There!
Server Recieved Message: Yt}}~1Eytct!
```

When the generated key was an arbitrary amount of 0's, both the man in the middle and server were able to read the message. In this specific case, the XOR cipher doesn't encrypt the data at all:

```
Server Key: 00
Middle Man Key: 00
Client Key: 00
Client Sent Message: Hello There!
MiddleMan Recieved Message: Hello There!
Server Recieved Message: Hello There!
```

In the case of this attack, the size of the Qubit stream does help improve security as there are more randomised bits, which will decrease the likelihood of a key match.

A better way to implement the middle man attack would have been to create the middle man with two sets of QKE connections. This means the client and middleman share a secret key whilst the server and middleman share another. That way the middleman can intercept the data and re-encode it without the recipient or sender knowing.

2.3 Defense

If separate keys are generated between the client and middleman, and the server and middleman, the length of the Qubit stream will not improve the defense of the attack. Using an asymetrical form of encryption will improve the security of the exchange, as it allows the client and the server to have a shared secret.