CS610 - Project 1: QKD-Encrypted LLM

Abraham J. Reines

September 15, 2024

Contents

1 Introduction

This project deploys a small pre-trained language model (GPT-2) on the ‘stu.cs.jmu.edu‘ server. It also uses Quantum Key Distribution (QKD) to encrypt communications between any client device and server (Patel et al., 2020; Yin et al., 2017). The goal is to explore quantum cryptographic techniques and their application in modern systems for secure communications, aligning with the course’s focus on networking and security (Tanenbaum et al., 2020).

2 High-Level Design

2.1 System Architecture

The system is designed as a client-server architecture. On the server side, GPT-2 is hosted via the Hugging Face ‘transformers‘ library to handle natural language processing tasks (Alammar, 2024; Face, 2024a).

- **Server Side:** The server hosts the pre-trained LLM (GPT-2) and handles encrypted communication with clients. It runs on ‘stu.cs.jmu.edu‘ (Lo et al., 2012).
- **Client Side:** Clients communicate with the server over a secure network. They use QKD to exchange encryption keys, which are used to encrypt and decrypt communications (Patel et al., 2020; Scarani & Kurtsiefer, 2014).

2.2 Quantum Key Distribution (QKD)

A QKD module is implemented to securely exchange encryption keys between the client and server. This module leverages cryptographic protocols described in (Agency, 2024; Bennett & Brassard, 1984; Lo et al., 2012; Scarani & Kurtsiefer, 2014).

2.3 Encryption and Decryption

- The LLM model is stored on the server and is accessed by clients via encrypted communications.
- When a client connects, the QKD process is initiated to securely exchange the encryption key, which is used to encrypt and decrypt the data sent between the client and server (Nielsen & Chuang, 2010; Patel et al., 2020).
- The model responses are encrypted before transmission to ensure secure communications. This is particularly crucial when using pre-trained models like GPT-2 over open networks (Face, 2024c; Labonne, 2024).

3 Implementation Details

3.1 Programming Language

The project is implemented in Python, which is supported on ‘stu.cs.jmu.edu‘. Python’s ecosystem provides excellent libraries for both machine learning (e.g., Hugging Face ‘transformers‘ for GPT-2) and cryptography (e.g., Qiskit for quantum encryption) (Face, 2024c; Nielsen & Chuang, 2010; Tanenbaum et al., 2020).

3.2 Specialized Libraries

- **Hugging Face Transformers:** Used to deploy the GPT-2 model for text generation tasks on the server. Hugging Face provides an intuitive API for integrating natural language processing (NLP) models such as GPT-2. It allows scientists to fine-tune and utilize pre-trained language models, perfect for designing an ultra secure cybersecurity knowledge base(Alammar, 2024; Face, 2024b).
- **Qiskit:** Qiskit is an open-source quantum computing framework developed by IBM, enabling researchers to work on quantum algorithms. Qiskit is employed for encryption and decryption processes, through principles of quantum key distribution (QKD). QKD leverages the fundamental laws of quantum mechanics (more on this later) to securely share cryptographic keys between two parties. This method is considered highly secure compared to classical cryptographic methods (Agency, 2024; Bennett & Brassard, 1984; Lo et al., 2012).

This library will be crucial ’training wheels’ for developing a robust QKD framework:

```python
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
import numpy as np

class QuantumProcessor:
    """
    QuantumProcessor simulates the BB84 Quantum Key Distribution (QKD)
    ↪ protocol.
    It handles quantum state preparation, measurement, key generation, and
    ↪ message encryption/decryption.
    """

    def __init__(self):
        """
        Initialize the QuantumProcessor with a quantum simulator and empty
    ↪ shared key.
        """
        self.shared_key = None
        self.simulator = AerSimulator()

    def prepare_quantum_state(self, bit, basis):
        """
        Prepare the quantum state based on the given bit and basis.

```

```
        :param bit: The classical bit (0 or 1) to encode into the quantum state.
        :param basis: The basis to use for encoding ('Z' for rectilinear or 'X' for diagonal).
        :return: A QuantumCircuit object representing the prepared quantum state.
        """
        qc = QuantumCircuit(1, 1)
        if bit == '1':
            qc.x(0)  # Apply X gate to encode bit 1
        if basis == 'X':
            qc.h(0)  # Apply H gate for diagonal basis
        return qc

    def measure_quantum_state(self, qc, basis):
        """
        Measure the quantum state based on the given measurement basis.

        :param qc: The QuantumCircuit containing the quantum state.
        :param basis: The basis to measure in ('Z' for rectilinear or 'X' for diagonal).
        :return: The measured classical bit (0 or 1).
        """
        if basis == 'X':
            qc.h(0)  # Apply H gate to switch to diagonal basis for measurement
        qc.measure(0, 0)
        job = transpile(qc, self.simulator)
        result = self.simulator.run(job, shots=1).result()
        counts = result.get_counts()
        return '1' if '1' in counts else '0'

    def generate_shared_key(self, alice_bits, alice_bases, bob_bases, bob_results):
        """
        Generate a shared key using the BB84 protocol by comparing Alice's and Bob's bases.

        :param alice_bits: The bits prepared by Alice.
        :param alice_bases: The bases used by Alice to encode the bits.
        :param bob_bases: The bases used by Bob to measure the qubits.
        :param bob_results: The measurement results obtained by Bob.
        :return: The sifted shared key.
        """
        sifted_key = [
            alice_bit for alice_bit, alice_base, bob_base, bob_result
            in zip(alice_bits, alice_bases, bob_bases, bob_results)
            if alice_base == bob_base  # Only keep the bits where bases match
        ]
```

```python
        self.shared_key = ''.join(sifted_key)
        print(f"Shared key generated: {self.shared_key}")  # Debugging line
        return self.shared_key

    def encrypt_message(self, message):
        """
        Encrypt a message using the shared key.

        :param message: The plaintext message to be encrypted.
        :return: The encrypted message as a string.
        :raises ValueError: If the shared key is not set.
        """
        if self.shared_key is None:
            raise ValueError("Shared key is not initialized.")

        # Encrypt using XOR with the shared key
        encrypted_message = ''.join(
            chr(ord(c) ^ int(self.shared_key[i % len(self.shared_key)], 2))
            for i, c in enumerate(message)
        )
        print(f"Encrypted message: {encrypted_message}")  # Debugging line
        return encrypted_message

    def decrypt_message(self, encrypted_message):
        """
        Decrypt an encrypted message using the shared key.

        :param encrypted_message: The encrypted message to be decrypted.
        :return: The decrypted message.
        :raises ValueError: If the shared key is not set.
        """
        if self.shared_key is None:
            raise ValueError("Shared key is not initialized.")

        # Decrypt by reapplying XOR (symmetric decryption)
        decrypted_message = self.encrypt_message(encrypted_message)  # XOR
    ↪ decryption
        print(f"Decrypted message: {decrypted_message}")  # Debugging line
        return decrypted_message

    def cleanup(self):
        """
        Clean up resources by deleting the quantum simulator.
        """
        print("Cleaning up simulator resources.")  # Debugging line
        del self.simulator
```

Listing 1: QKD Module

```python
from transformers import pipeline
import gc
```

```python


class GPT2Model:
    """
    GPT2Model is responsible for generating text responses using the GPT-2 model.
    It initializes the model, generates responses based on a given prompt, and cleans up resources.
    """

    def __init__(self):
        """
        Initialize the GPT2Model with the GPT-2 text generation pipeline.
        The model is set to use the CPU.
        """
        self.model = pipeline(
            "text-generation", model="gpt2", device=-1
        )  # Set device to CPU

    def generate_response(
        self, prompt, max_length=50, num_return_sequences=1, temperature=1.0
    ):
        """
        Generate a text response based on the given prompt.

        :param prompt: The input text prompt for the model.
        :param max_length: The maximum length of the generated text.
        :param num_return_sequences: The number of generated sequences to return.
        :param temperature: The sampling temperature for text generation.
        :return: The generated text response.
        """
        try:
            return self.model(
                prompt,
                max_length=max_length,
                num_return_sequences=num_return_sequences,
                temperature=temperature,
                pad_token_id=self.model.tokenizer.eos_token_id,
                clean_up_tokenization_spaces=True,
            )[0]["generated_text"]
        except Exception as e:
            print(f"An error occurred during text generation: {e}")
            return ""

    def cleanup(self):
        """
        Clean up resources by deleting the model and performing garbage collection.
```

```python
        """
        del self.model
        gc.collect()


if __name__ == "__main__":
    gpt2 = GPT2Model()
    prompt = "Once upon a time"
    response = gpt2.generate_response(prompt)
    print(response)
    gpt2.cleanup()
```

Listing 2: LLM Module

```python
import requests
from qkd import QuantumProcessor


class QKDClient:
    """
    QKDClient is responsible for initiating the Quantum Key Distribution (QKD) process with the server,
    encrypting messages using the shared key, and sending encrypted messages to the server.
    """

    def __init__(self, server_url="http://0.0.0.0:8000/qkd"):
        """
        Initialize the QKDClient with the server URL and create an instance of QuantumProcessor.

        :param server_url: The URL of the server to initiate QKD with.
        """
        self.server_url = server_url
        self.shared_key = None
        self.qkd = QuantumProcessor()
        print("QKDClient initialized with server URL:", self.server_url)

    def initiate_qkd(self):
        """
        Initiate the QKD process by preparing the quantum state, measuring it, and sending the results to the server.
        The server responds with its own measurement results, which are used to generate a shared key.
        """
        print("Preparing quantum state...")
        alice_bits = ['0', '1', '0', '1']  # Example bits
        alice_bases = ['Z', 'X', 'Z', 'X']  # Example bases

        for bit, basis in zip(alice_bits, alice_bases):
            self.qkd.prepare_quantum_state(bit, basis)
```

```python

        alice_measurements = [self.qkd.measure_quantum_state(self.qkd.prepare_quantum_state(bit, basis), basis) for bit, basis in zip(alice_bits, alice_bases)]
        print("Sending QKD initiation request to server...")
        try:
            response = requests.post(
                self.server_url,
                json={"alice_bits": alice_bits, "alice_bases": alice_bases},
            )
            print(f"Server response content: {response.content}")  # Debugging line

            if response.status_code == 200:
                try:
                    data = response.json()
                    print("Generating shared key...")
                    self.shared_key = self.qkd.generate_shared_key(
                        alice_bits, alice_bases, data["bob_bases"], data["bob_results"]
                    )
                    print("Shared key generated:", self.shared_key)
                except requests.exceptions.JSONDecodeError as e:
                    print(f"Error decoding JSON response: {e}")
            else:
                print(f"Server returned an error: {response.status_code}")
        except requests.exceptions.RequestException as e:
            print(f"Error connecting to server: {e}")

    def encrypt_message(self, message):
        """
        Encrypt a message using the shared key.

        :param message: The plaintext message to be encrypted.
        :return: The encrypted message, or None if the shared key is not initialized.
        """
        if self.shared_key is None:
            print("Error: Shared key is not initialized.")
            return None

        print("Encrypting message...")
        encrypted_message = "".join(
            chr(ord(c) ^ int(self.shared_key[i % len(self.shared_key)], 2))
            for i, c in enumerate(message)
        )
        print("Message encrypted.")
        return encrypted_message

```

```python
    def send_encrypted_message(self, message):
        """
        Send an encrypted message to the server and decrypt the server's response.

        :param message: The plaintext message to be encrypted and sent.
        :return: The decrypted response from the server, or None if an error occurs.
        """
        print("Sending encrypted message to server...")
        encrypted_message = self.encrypt_message(message)
        if encrypted_message is None:
            print(
                "Error: Could not encrypt message because the shared key is not initialized."
            )
            return None

        try:
            response = requests.post(
                f"{self.server_url.replace('/qkd', '')}/generate",
                json={"text": encrypted_message},
            )
            print(f"Raw server response: {response.content}")  # Debugging line
            encrypted_response = response.json().get("response", "")
            return self.qkd.decrypt_message(encrypted_response)  # Decrypt the response
        except requests.exceptions.RequestException as e:
            print(f"Error sending encrypted message: {e}")
            return None


if __name__ == "__main__":
    client = QKDClient()
    client.initiate_qkd()

    message = "Once upon a time... "
    print("Sending message:", message)
    decrypted_response = client.send_encrypted_message(message)
    print(f"Server Response: {decrypted_response}")
```
Listing 3: Client Module

```python
import os

os.environ["TOKENIZERS_PARALLELISM"] = "false"  # Disable tokenizers parallelism

from flask import Flask, request, jsonify
from asgiref.wsgi import WsgiToAsgi
```

```python
import warnings
import atexit
import numpy as np

from model import GPT2Model
from qkd import QuantumProcessor

# Suppress the FutureWarning
warnings.filterwarnings(
    "ignore", category=FutureWarning, module="transformers.tokenization_utils_base"
)

app = Flask(__name__)  # Expose app at the module level

# Instantiate  server components at the module level
model = GPT2Model()
qkd = QuantumProcessor()


@app.route("/qkd", methods=["POST"])
def qkd_exchange():
    data = request.json
    print(f"Received data: {data}")  # Debugging line
    alice_bits = data.get("alice_bits")
    alice_bases = data.get("alice_bases")

    # Simulate Bob's process
    bob_bases = ["Z" if np.random.rand() > 0.5 else "X" for _ in alice_bits]
    bob_results = [
        qkd.measure_quantum_state(qkd.prepare_quantum_state(bit, basis), basis)
        for bit, basis in zip(alice_bits, bob_bases)
    ]

    qkd.shared_key = qkd.generate_shared_key(
        alice_bits, alice_bases, bob_bases, bob_results
    )
    print(f"Shared key set: {qkd.shared_key}")  # Debugging line
    response = {"bob_bases": bob_bases, "bob_results": bob_results}
    print(f"Response data: {response}")  # Debugging line
    return jsonify(response)


@app.route("/generate", methods=["POST"])
def generate_response():
    data = request.json
    encrypted_text = data.get("text", "")
    print(f"Encrypted text received: {encrypted_text}")  # Debugging line
```

```python
    print(f"Shared key before decryption: {qkd.shared_key}")  # Debugging line
    decrypted_text = qkd.decrypt_message(
        encrypted_text
    )  # Decrypt using shared key logic
    print(f"Decrypted text: {decrypted_text}")  # Debugging line
    if decrypted_text is None:
        return jsonify({"error": "Decryption failed"}), 400
    response_text = model.generate_response(decrypted_text)
    print(f"Response text: {response_text}")  # Debugging line
    if response_text is None:
        return jsonify({"error": "Model response generation failed"}), 500
    encrypted_response = qkd.encrypt_message(
        response_text
    )  # Encrypt using shared key logic
    print(f"Encrypted response: {encrypted_response}")  # Debugging line
    return jsonify({"response": encrypted_response})


def cleanup_resources():
    model.cleanup()
    qkd.cleanup()


atexit.register(cleanup_resources)

if __name__ == "__main__":
    asgi_app = WsgiToAsgi(app)
    import uvicorn

    uvicorn.run(asgi_app, host="0.0.0.0", port=8000)
```

Listing 4: Server Module

4 Compiling and Running the Project

4.1 Setup Instructions

- Ensure Python 3.10 is installed on both the server and client machines using pyenv (Nielsen & Chuang, 2010).
- Create and activate a virtual environment on both the server and client:

```
python3 -m venv QKD-LLM_env
source QKD-LLM_env/bin/activate
```

- Install the required libraries using the ‘requirements.txt‘ file (Tanenbaum et al., 2020):

```
pip install -r requirements.txt
```

- Ensure ‘server.py‘, ‘client.py‘, and ‘qkd.py‘ are correctly placed in their respective directories on the server and client machines (Lo et al., 2012).

4.2 Running the Server

To start the server on the ‘stu.cs.jmu.edu‘ server, run:

```
python3 server.py
```

4.3 Running the Client

To start the client on a local or remote machine, run:

```
python3 client.py
```

5 Integration with Qiskit

5.1 QKD Simulation with Qiskit

The Proof of Concept (PoC) currently simulates QKD using Python. However, a future version of the project will use Qiskit to perform quantum key exchanges on real quantum hardware provided by IBM (**nsa_2024**). Qiskit will simulate the quantum entanglement and measurement required for secure key exchange:

- **Qubit preparation and measurement** will be handled by Qiskit, using the BB84 protocol (Bennett & Brassard, 1984).
- **Quantum simulation**: Qiskit's Aer simulator can simulate noisy quantum channels, including photon loss or interference which would occur in a real quantum environment (Nielsen & Chuang, 2010).
- **Integration**: Once the quantum key is generated using Qiskit, it will be passed into the encryption process of the LLM communications.

6 Quantum Mechanics of Photon Transmission

Quantum Key Distribution (QKD) relies on the principles of quantum mechanics to ensure the security of key exchanges between two parties, typically referred to as Alice (sender) and Bob (receiver) (Patel et al., 2020). The foundation of QKD lies in the transmission of qubits, which in many implementations are encoded in the states of photons (Lo et al., 2012).

6.1 Photon as a Qubit Carrier

In QKD protocols such as BB84, photons serve as carriers for qubits (Bennett & Brassard, 1984). A photon's quantum state can be used to encode binary information, through its polarization:

- **Polarization states:** A photon can be polarized horizontally ($|0\rangle$) or vertically ($|1\rangle$), representing classical bit values. Additionally, diagonal polarizations ($|+\rangle$, $|-\rangle$) are used to create superposition states existing as both 1 and 0 simultaneously (Nielsen & Chuang, 2010). This is what makes quantum computers so fast.

- **Quantum superposition:** A single photon can exist in a superposition of $|0\rangle$ and $|1\rangle$, meaning its state is not fixed until measured. Mathematically, this is described by the quantum state:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

  where α and β are complex probability amplitudes which satisfy $|\alpha|^2 + |\beta|^2 = 1$. This property allows for encoding information in both bases (rectilinear and diagonal) in QKD protocols (Nielsen & Chuang, 2010).

6.2 Quantum Superposition and Measurement

The principle of superposition ensures a photon's polarization exists in a probability distribution of states (Yin et al., 2017). Upon measurement, the quantum state collapses to one of the possible outcomes:

- If measured in the rectilinear basis (horizontal/vertical), the photon collapses to either $|0\rangle$ or $|1\rangle$ with probabilities corresponding to its superposition (Patel et al., 2020).

- If measured in the diagonal basis, the photon collapses to $|+\rangle$ or $|-\rangle$, again based on its quantum state before measurement (Nielsen & Chuang, 2010).

This probabilistic behavior is key to detecting potential eavesdropping: if an eavesdropper (Eve) tries to measure the photons in transit, the measurement will be corrupted (due to the collapsed quantum state) and detected in the key generation process (Scarani & Kurtsiefer, 2014).

6.3 Transmission through a Quantum Channel

Photons are transmitted through a quantum channel, which can be either a fiber-optic cable or free space (radio frequency transmission via photons). The transmission medium introduces challenges such as attenuation, noise, and loss of photon coherence (Patel et al., 2020). During transmission:

- **Photon loss:** Some photons may be absorbed or scattered by the medium (atmosphere), which reduces the number of qubits which reach the receiver(Aktas, Wengerowsky, et al., 2016).

- **Decoherence:** A photon's quantum state may change unpredictably due to environmental interactions, especially over long distances (Nielsen & Chuang, 2010).

6.4 Eavesdropping Detection via the No-Cloning Theorem

The **no-cloning theorem**, states an unknown quantum state cannot be copied (Lo et al., 2012). If an eavesdropper attempts to intercept the photon, they will alter its state due to the act of measurement. (Bennett & Brassard, 1984).

6.5 Quantum Key Agreement and Security

Once the photons have been transmitted and measured, Alice (client) and Bob (server) compare their measurement bases. Bits where both used the same basis (either rectilinear or diagonal) are kept for the key (Bennett & Brassard, 1984). Any attempt to eavesdrop the qubits will introduce errors, which Alice and Bob can detect by comparing their key (Lo et al., 2012).

7 Language Model (LLM) on the Server

The language model (LLM) deployed on the server is a pre-trained instance of **GPT-2**, a transformer-based model from the Hugging Face library (Face, 2024a). GPT-2 is designed to predict the next token in its context using the vectors of matrices formulated by GPTs training corpus, using basic linear algebra. In the current implementation, this LLM facilitates communication by responding to client queries with natural language responses. This model is prone to profanity and nonsense. It serves as a placeholder.

7.1 Future Upgrades to Ollama LLM

In future iterations of this project, the current GPT-2 model will be replaced by the latest **Ollama LLM** (Face, 2024b), which offers advanced capabilities and a larger model size, resulting in more sophisticated language generation. Alongside this upgrade, a complex **llm-pipeline-engine** may be implemented for sophisticated error handling. This upgrade will enhance the system's ability to generate context-aware responses, making it suitable for more complex applications such as decision support, intelligence analysis, and secure communication.

7.2 Potential for Fine-Tuning

The current implementation uses a pre-trained version of GPT-2 without any additional training or fine-tuning (Alammar, 2024). However, future versions of the system could integrate **fine-tuning** capabilities to further customize the LLM for specific tasks or industries:

- Fine-tuning could involve training the LLM on industry specific data, such as government or military documents, to improve the model's performance on specialized queries. For example, classified technical schematics for naval equipment.
- The model can also be fine-tuned to optimize its response generation for sensitive or mission-critical communications (Face, 2024b).

7.3 Server-Client Architecture

In the current server-client architecture, the language model operates as follows (Labonne, 2024):

- **Server Side:** The server hosts the GPT-2 model, managing client requests and generating responses based on the inputs it receives. After the secure quantum key exchange via QKD, the server uses the shared encryption key to decrypt incoming client requests.
- **Client Side:** Clients send encrypted text queries to the server. Once the server decrypts the query using the key established by the QKD process, the GPT-2 model processes the request and generates a text response. This response is then encrypted with the same quantum key and sent back to the client (Face, 2024a).
- **Encryption Workflow:** Both client queries and server responses are encrypted to maintain the confidentiality of communications (SemaphoreCI, 2024).

7.4 Response Generation and Latency Considerations

Upon receiving an encrypted query from the client, the server decrypts the input and passes it to the LLM. GPT-2 generates a natural language response, which is then encrypted and transmitted back to the client (Face, 2024a).

8 Known Issues

- The QKD is currently simulated, and the Qiskit integration remains untested (Agency, 2024). There is currently an error with basic encryption and decryption.
- The decryption of the model responses may introduce latency, depending on the computational power of the client machine and network conditions (Face, 2024a).
- Additional performance testing is needed to evaluate the impact of QKD on communication speed.

9 Conclusion

This project successfully integrates quantum cryptographic techniques with machine learning, offering a secure approach to deploying LLMs in a networked environment. By incorporating Quantum Key Distribution (QKD), the system is protected against quantum computing threats (Agency, 2024). In future versions, incorporating **Qiskit** will enhance quantum-based security by using real quantum devices for QKD (Nielsen & Chuang, 2010).

The potential applications of this system extend to fields where secure communications are critical, such as government and military operations, where the security of information could impact national security (Agency, 2024).

References

Agency, N. S. (2024). Quantum key distribution (qkd) and quantum cryptography (qc) [Accessed: 2024-09-15]. https://www.nsa.gov/Cybersecurity/Quantum-Key-Distribution

Aktas, D., Wengerowsky, S. et al. (2016). Quantum cryptography for global secure communication. *Laser Photon Reviews*, *10*(5), 451–457. https://doi.org/10.1002/lpor.201600059

Alammar, J. (2024). The illustrated gpt-2: Visual introduction to gpt models [Accessed: 2024-09-15]. https://jalammar.github.io/illustrated-gpt2/

Bennett, C. H., & Brassard, G. (1984). Quantum cryptography: Public key distribution and coin tossing. *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, 175–179. https://doi.org/10.1109/ICCSSP.1984.1299395

Face, H. (2024a). Gpt-2 by hugging face: Documentation and usage [Accessed: 2024-09-15]. https://huggingface.co/gpt2

Face, H. (2024b). Open-source llms and text generation models on hugging face hub [Accessed: 2024-09-15]. https://huggingface.co/docs

Face, H. (2024c). Openai gpt-2 model page [Accessed: 2024-09-15]. https://huggingface.co/openai-gpt2

Labonne, M. (2024). Llm course with roadmaps and colab notebooks [Accessed: 2024-09-15]. https://github.com/mlabonne/llm-course

Lo, H.-K., Curty, M., & Qi, B. (2012). Measurement-device-independent quantum key distribution. *Physical Review Letters*, *108*(13), 130503. https://doi.org/10.1103/PhysRevLett.108.130503

Nielsen, M. A., & Chuang, I. L. (2010). *Quantum computation and quantum information: 10th anniversary edition*. Cambridge University Press.

Patel, K. A., Dynes, J. F., & Choi, I. (2020). Coexistence of high-bit-rate quantum key distribution and data on optical fiber. *Physical Review Applied*, *6*(1), 054022. https://doi.org/10.1103/PhysRevApplied.6.054022

Scarani, V., & Kurtsiefer, C. (2014). The black paper of quantum cryptography: Real implementation problems. *Theoretical Computer Science*, *560*, 27–32. https://doi.org/10.1016/j.tcs.2014.10.001

SemaphoreCI. (2024). 6 ways to run llms locally: Hugging face and alternatives [Accessed: 2024-09-15]. https://semaphoreci.com/blog/run-llms-locally

Tanenbaum, A. S., Feamster, N., & Wetherall, D. (2020). *Computer networks (6th ed.)* Pearson.

Yin, J., Cao, Y., Li, Y. H., et al. (2017). Satellite-based entanglement distribution over 1200 kilometers. *Science*, *356*(6343), 1140–1144. https://doi.org/10.1126/science.aan3211

Academic Integrity Pledge

“This work complies with JMU honor code. I did not give or receive unauthorized help on this assignment.”