Applying Bernstein-Vazirani Algorithm to Cryptography

Maria Harrison
CS 8395-53: Advanced Quantum Computing
Vanderbilt University
Nashville, TN, USA
maria.harrison@vanderbilt.edu

***Abstract*—The purpose of this paper is to explore the Bernstein-Vazirani algorithm, and gain a better understanding of its application in cryptography.**

Keywords—oracle, quantum states, hidden shift, speed up, key encryption.

I. Introduction

In 1992, Ethan Bernstein and Umesh Vazirani developed the Bernstein-Vazirani algorithm to solve a hidden shift problem. This algorithm has numerous applications in cryptography, error correction, and optimization problems. The goal of this problem was to find a hidden shift.

II. Background

The intention of the Bernstein-Vazirani algorithm is to utilize quantum mechanics's superposition and entanglement properties to efficiently determine an unknown string. This algorithm identifies a hidden binary string in one single run. In comparison to classical algorithms, this provides significant speed up times. The Bernstein-Vazirani algorithm provides a quantum algorithm with linear speedup in the amount of queries relative to the best classical algorithm.

A. Hidden Shift Problem

The hidden shift problem, also known as the Bernstein-Vazirani problem, states that when given a function there is a guarantee an undetermined shift exists; An unknown bit string exists. The function is guaranteed to be of type $a \cdot x$ and when given an input, the output will always be of $a \cdot x\ (mod 2)$.

When f is a given function of bit strings and
a is the phase shift
x is a string of n bits
n is the length of said bit string $\{x_0, x_1, x_2, \ldots, x_n\}$
The algorithm defines a function $f(x)$ to determine the unknown n-bit string a as followed:
$f(x) = a \cdot x\ (mod 2)$

B. The Inner-Product Oracle

This algorithm uses a quantum oracle that applies a given function to a superposition of all possible inputs of a quantum state. Often referred to as the inner-product oracle, the quantum oracle is an inner modulus function that receives queries and applies a single-qubit unitary transformation. The complexity of the Bernstein-Vazirani algorithm is in terms of the number of queries to an oracle.

C. The Quantum Circuit

This algorithm uses n qubits, and outputs each bit of the hidden string after measurements are made. This process solves the Berstein-Vazirani problem in one query, rather than running multiple passes, at minimum of n runs for a string of length n. This provides significant linear speed up.

III. Applications in Cryptography

Encryption, in the context of cryptography, is a process in which plain text or a piece of information is converted into cipher text that can only be decoded by the receiver for whom the information is intended. A cryptographic key is a string of randomized characters, often mathematically generated, paired with a cryptographic algorithm to secure data in a message. There are two general categories of cryptographic keys, symmetric and asymmetric encryption. There are four encryption algorithms, triple DES, RSA, Twofish, and AES.

A. Symmetric and Asymmetric Key Encryption

Symmetric key encryption encrypts and decrypts a message using the same key. While this process is very fast, it only provides confidentiality. The length of the key used is either 128 or 256 bits.

Asymmetric key encryption is based on public and private key encryption techniques; one key is used for encryption only and the other for decryption. This process is slower than symmetric, but provides confidentiality, authenticity, and non-repudiation. The length of the key used is 2048 bits or larger.

B. Cracking Cryptographic Key Encryptions

Cryptographic keys vary in length, spanning from 128 to 256 bits, to 2046 bits. The runtime of classical algorithms significantly limits its applications due to its minimum runtime of n runs, for a string of n length. If a classical algorithm was used to crack a cryptographic key of a 128 bit size, the runtime would be a minimum of 128 runs, just to fully traverse the key in its entirety. Bernstein-Vazirani's algorithm is highly applicable to cryptographic key-related attacks because of its significant speedup compared to classical algorithms.

IV. Exploration of Bernstein-Vazirani

To further understand Bernstein-Vazirani's algorithm, I implemented it using quantum circuits, and used it to guess a Fernet encryption key. My coding environment consisted of:

- Visual Studio Code as an IDE
- Python 3.11 language
- Qiskit, Numpy, Pyenv Python submodules

A. Setting Up for a Quantum Circuit

First I generated a random int, in decimal type, whose binary format would be used as my hidden string.

Next, I generated a random int between 1, 39 which represented the number of qubits for my circuit.

Then, I performed a modulus function on my hidden string, checking to ensure that my randomly generated integer would be able to be represented with the randomly generated number of qubits.

When f is my inner product oracle function and

r is a randomly generated integer used for my hidden string

n_qubits is a randomly generated integer for the number of qubits my circuit will use

$secret\_num$ is the inner product from a modulus function

The hidden string, number of qubits for my quantum circuit is as followed:

```
def getRandomNumberHigh():
    decimal_num = random.randint(1, 1000)
    print("Random number HIGH generated in decimal format:", decimal_num)
    print("Random number in BINARY format:", bin(decimal_num).replace("0b", ""))
    return decimal_num

def getRandomNumberQubits():
    n_qub = random.randint(1, 39)
    print("Random number of qubits:", n_qub)
    return n_qub
```

0.0s

```
r = getRandomNumberHigh()
n_qubits = getRandomNumberQubits()
```

0.0s

```
Random number HIGH generated in decimal format: 864
Random number in BINARY format: 1101100000
Random number of qubits: 12
```

```
# sanitation check that the hidden number can be represented with n number of qubits
secret_num = r %2**(n_qubits)
```

0.0s

B. Creating Bernstein-Vazirani Algorithm

When $bernstein\_vazirani\_alg(r, n)$ is the algorithm and

r is the hidden string in decimal format

n is the number of qubits for the quantum circuit

qr is a quantum register of size $n + 1$

cr is a classical register of size n

$bvCirq$ is a QuantumCircuit

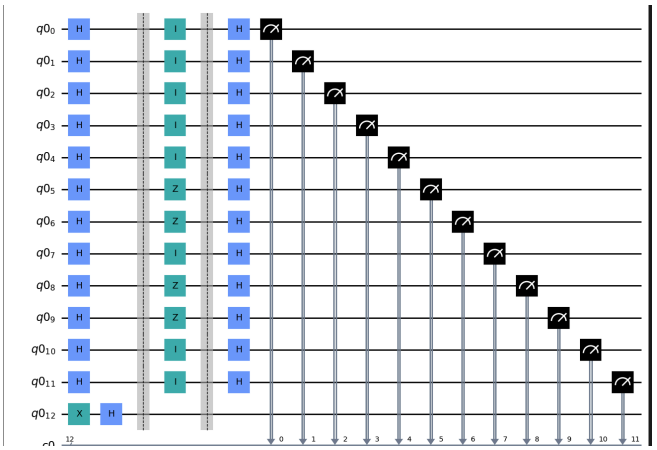
$.x$ is the output $x\ gate$

$.h$ is the output $h\ gate$

$if\ (f\ \&\ 1\ <<\ i))$: is the inner product oracle

The newly created quantum circuit implementing Bernstein-Vazirani's algorithm is as followed:

```
'''
def bernstein_vazirani_alg(r, n):
    qr = QuantumRegister(n+1)
    cr = ClassicalRegister(n)
    bvCirq = QuantumCircuit(qr, cr, name ="BernsteinVazirani")

    # setting output x gate qubits
    bvCirq.x(n)

    # setting output h gate qubits
    bvCirq.h(n)

    # set up input register and setting h gate
    for qubit in range(n):
        bvCirq.h(qubit)

    bvCirq.barrier()

    # add return from oracle() func to circuit
    # bvCirq.append(oracle, range(n+1))
    for i in range(n):
        if (r & (1 <<i)):
            bvCirq.z(qr[i])
        else:
            bvCirq.id(qr[i])

    bvCirq.barrier()

    # set h gates again before measuring
    for q in range(n):
        bvCirq.h(qr[q])

    # measure perforamnce
    for i in range(n):
        bvCirq.measure(i, i)

    return bvCirq
```



C. Creating Quantum Simulator and Executing a Circuit

Next I created a quantum simulator and executed my newly created quantum circuit, implemented with Bernstein-Vazirani's Algorithm.

```
'''
    Running Bernstein Vazirani Algorithm
'''
# creating quantum simulator
simulator = Aer.get_backend('qasm_simulator')
shots = 100
# running quantum circuit on simulator and calling .result() to get results of circuit when it runs
result = execute(qCirc, backend=simulator, shots=shots).result()

#printing histogram result
plot_histogram(result.get_counts(qCirc))
```

D. Measuring and Testing Algorithm

Next I measured and tested my algorithm, and compared the hidden string with the algorithm's guess to determine

accuracy.

```
guess = list(result.get_counts().keys())
print(guess)
✓ 0.0s

['001101100000']
```

```
def isGuessCorrect(secret_num, guess):
    if (format(guess == secret_num)):
        print("Yay, the guess was CORRECT!")
    else:
        print("Sorry, the guess was NOT correct :(")
✓ 0.0s
```

```
def printOutputAndScore(r, guess, secret_num):
    print("Random number generated at the beginning and converted to secret string :", r)
    print("Secret string:", secret_num)
    print("Guess:", guess)

    isGuessCorrect(secret_num, guess)

printOutputAndScore(r, guess, secret_num)
✓ 0.0s

Random number generated at the beginning and converted to secret string : 864
Secret string: 864
Guess: ['001101100000']
Yay, the guess was CORRECT!
```

V. Applying Bernstein-Vazirani to Guess a Fernet Encryption Key

To further explore the application of Bernstein-Vazirani in cracking cryptographic keys, I simulated a very elementary level encryption message program.

A. Setting Up

To best replicate and demonstrate the difficulty and randomness of key encryption algorithms, I opted to create a completely random message using a random sentence generator. I utilized Python's Wonderword module to generate 4 complete sentences at random, with the criteria of a noun, verb, adjective, and direct object. I chose a random sentence generator to create my message because I wanted to test my algorithm to determine if it could succinctly behave with messages of varying length, which would result in encryption keys of different sizes.

```
count = 4
random_msg = generateMessage(count)
print(random_msg)
✓ 0.0s

The gamy beech realizes round.
The ossified cauliflower dries skunk.
The spicy bustle navigates sanction.
The good lacquerware assists divalent.
['The gamy beech realizes round.', 'The ossified cauliflower dries skunk.', 'The spicy bustle navigates sanction.', 'The good lacquerware assists divalent.']
```

```
message = ' '.join(random_msg)
print(message)
✓ 0.0s

The gamy beech realizes round. The ossified cauliflower dries skunk. The spicy bustle navigates sanction. The good lacquerware assists divalent.
```

```
def printEncryption(original, encrypt, decrypt):
    print("Original message before encryption")
    print("----------------------------------")
    print(original)
    print("\n")
    print("Encrypted message after encrypting")
    print("----------------------------------")
    print(encrypt)
```

B. Encrypting a Random Message

Next, I encrypted my randomly generated message using Fernet and then continued the process of decoding the encryption key to use as my hidden string in the algorithm.

```
print("Fernet key encrypted: ", fernet)
✓ 0.0s

Fernet key encrypted:  b'n1QbDGJxjXLzv5od6XZEi0nQN9CrJAqBhPbNqYLe3ys='
```

```
'''
    Encrypting Message using Fernet
'''
def fernet_encryption(message):
    #generate key for encryption using fernet
    fernet_key = Fernet.generate_key()

    fernet = Fernet(fernet_key)

    #encrypt the string, but message has to be encoded to byte string first
    fern_encrypt = fernet.encrypt(message.encode())
    fern_decrypt = fernet.decrypt(fern_encrypt).decode()
    print("Fernet Encryption")
    print("################")
    print("\n")
    printEncryption(message, fern_encrypt, fern_decrypt)
    return fernet_key
✓ 0.0s                                                    Python

fernet = fernet_encryption(message)
✓ 0.0s                                                    Python
```

```
Fernet Encryption
################


Original message before encryption
----------------------------------
The gamy beech realizes round. The ossified cauliflower dries skunk. The spicy bustle navigates sanction. The good lacquerware assists divalent.


Encrypted message after encrypting
----------------------------------
b'gAAAAABlarFQ5U6srQgi9uUMl2c_NGEb2kdyTwlSuKklhP0EQAPYG8B_hD5BwqG3v8uT92NVVLNcYtM6iuBfeGJuGgRs7CLYALTagt_ltjTqUD_-0qW-t7HTZJ4dZ44fd8HoXSqBHe6mjb3mNYJDMqHcC0vyoMZWVNHTTwksoM9oyI7b4CZjMCpeib0Up5095r0znGvuJCjt1TOA2UnM


Decrypted message after decrypting
----------------------------------
The gamy beech realizes round. The ossified cauliflower dries skunk. The spicy bustle navigates sanction. The good lacquerware assists divalent.
```

```
import binascii

#fernet key is a base64 encoded 32byte key
print("Encrypted Key in Base64:")
print(fernet)
print("\n")

b64_ascii = fernet.decode("ascii")
print("Key of Base64 -> decoded in ASCII")
print(b64_ascii)
print("\n")

b64_binary = binascii.a2b_base64(b64_ascii)
print("Converting key of B64 -> Binary Array")
print(b64_binary)
print("\n")

print("Converting key of binary array -> bytes")
bits = ''.join(format(byte, '08b') for byte in b64_binary)
print(bits)
print("\n")

print("Converting key of binary bytes -> decimal")
converted =[b for b in b64_binary]
print(converted)
print("\n")

print("Sorted number of ints from converted Base64 key")
converted.sort()
print(converted)

✓ 0.0s
```

The following is the output of the encryption key after it has been decoded and converted into the proper format for my circuit. I opted to convert the final key from Base64 32 bytes to a byte array, and then transformed it into a list of integers. This was the best option for me to be able to randomly select an item from the list to be the hidden string for my algorithm.

```
Encrypted Key in Base64:
b'n1QbDGJxjXLzv5od6XZEi0nQN9CrJAqBhPbNqYLe3ys='


Key of Base64 -> decoded in ASCII
n1QbDGJxjXLzv5od6XZEi0nQN9CrJAqBhPbNqYLe3ys=


Converting key of B64 -> Binary Array
b'\x9fT\x1b\x0cbq\x8dr\xf3\xbf\x9a\x1d\xe9vD\x8bI\xd07\xd0\xab$\n\x81\x84\xf6\xcd\xa9\x82\xde\xdf+'


Converting key of binary array -> bytes
1001111101010100000110110000110001100010011100011000110101110010111100111011111110011010000111011110100101110110010001001000101101001001110100000011011111010000101010110010010000001010100000011000010011110110110011


Converting key of binary bytes -> decimal
[159, 84, 27, 12, 98, 113, 141, 114, 243, 191, 154, 29, 233, 118, 68, 139, 73, 208, 55, 208, 171, 36, 10, 129, 132, 246, 205, 169, 130, 222, 223, 43]


Sorted number of ints from converted Base64 key
[10, 12, 27, 29, 36, 43, 55, 68, 73, 84, 98, 113, 114, 118, 129, 130, 132, 139, 141, 154, 159, 169, 171, 191, 205, 208, 208, 222, 223, 233, 243, 246]
```

C. Creating a Simulator and Running Circuit

Next, I created a quantum simulator and executed my algorithm, using a random part of my decoded Fernet encryption key to serve as the hidden string. I really wanted to emphasize the randomness of the hidden string to demonstrate this algorithm's application capacity and runtime, hence why I opted to randomize a subset of my

decoded encryption key.

```
def getRandNumFernet(converted):
    rf = random.choice(converted)
    nqb = random.randint(3, 39)

    print("Random number from Fernet Base64 key in decimal format:", rf)
    print("Random number in BINARY format:", bin(rf).replace("0b", ""))
    print("Random number of qubits:", nqb)
    return rf, nqb
```

✓ 0.0s

```
# sanitation check that the hidden number can be represented with n number of qubits
rf, nqb = getRandNumFernet(converted)
if (rf%2 ** (nqb)):
    print("PASS")
    rf_secret = bin(rf).replace("0b", "")
    pass
else:
    print(rf%2 ** (nqb))
```

✓ 0.0s

```
Random number from Fernet Base64 key in decimal format: 84
Random number in BINARY format: 1010100
Random number of qubits: 15
PASS
```

D. *Measuring and Testing Algorithm*

Measuring and testing my algorithm follows the same steps in section IV, part D. The results can be seen as follows:

```
printOutputAndScore(rf, format(fernet_guess), rf_secret)
```

✓ 0.0s

```
Random number generated at the beginning and converted to secret string : 84
Secret string: 1010100
Guess: ['000000001010100']
Yay, the guess was CORRECT!
```

Lastly, after the algorithm finished running, I encoding the guessed string from the algorithm back into the original data encoding and format. I did this to demonstrate the capacity of applying Bernstein-Vazirani's algorithm in cryptography.

```
fernet_guess = list(fernet_result.get_counts().keys())
print(format(fernet_guess))
```

✓ 0.0s

```
['000000001010100']
```

```
def convertBackToBase64(guess):
    print("Fernet Guess in Binary")
    print(guess)
    print("\n")

    print("Converting Guess of binary bits -> decimal")
    bytes_bits = int(guess[0], 2)
    # bytes_bits = int(guess, 2)
    print(bytes_bits)
    print("\n")
```

✓ 0.0s

```
convertBackToBase64(fernet_guess)
```

✓ 0.0s

```
Fernet Guess in Binary
['000000001010100']


Converting Guess of binary bits -> decimal
84
```

VI. Concluding Remarks

Bernstein-Vazirani's Algorithm poses significant benefits when applied to cryptography. In my example application in this study, I demonstrated the algorithm's ability to identify two independent, randomly created hidden strings. Using a random binary string from a decimal integer as a hidden string was meant to help understand and build the foundations of the Bernstein-Vazirani algorithm. A random binary string, from a subset of a decoded Fernet encryption key demonstrated the capacity of applying this algorithm to crack other encryption keys, such as RSA. The algorithm exploits quantum circuits and querying an oracle, thus surpassing classical algorithms in its runtime alone. The use of querying an inner product oracle poses significant advantages as the algorithm is not required to run n times, in which n is the length of the hidden string. Creating a small scale cryptographic key application in this study barely scrapes the iceberg of cryptography, but provides insight to just one of the applications of the Bernstein-Vazirani algorithm.

VII. Works Cited

[1]
A. Shukla and P. Vedula, "A generalization of Bernstein–Vazirani algorithm with multiple secret keys and a probabilistic oracle," Quantum Inf Process, vol. 22, no. 6, p. 244, Jun. 2023, doi: 10.1007/s11128-023-03978-3.

[2]
K. Nagata, G. Resconi, T. Nakamura, J. Batle, S. Abdalla, and A. Farouk, "A Generalization of the Bernstein-Vazirani Algorithm," MOJ Ecology & Environmental Science, vol. 2, p. 00010, Mar. 2017, doi: 10.15406/mojes.2017.02.00010.

[3]
A. R. Khan, B. Rizwan, and F. Hassan, "Bernstein-Vazirani Algorithm".

[4]
"QasmBackendConfiguration," IBM Quantum Documentation. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.providers.models.QasmBackendConfiguration

[5]
B.-M. Zhou and Z. Yuan, "Quantum key-recovery attack on Feistel constructions: Bernstein–Vazirani meet Grover algorithm," Quantum Inf Process, vol. 20, no. 10, p. 330, Sep. 2021, doi: 10.1007/s11128-021-03256-0.

[6]
H. Xie and L. Yang, "Using Bernstein-Vazirani Algorithm to Attack Block Ciphers." arXiv, Jul. 16, 2018. [Online]. Available: http://arxiv.org/abs/1711.00853

[7]
"Wonderwords Official Documentation — Wonderwords documentation." [Online]. Available: https://wonderwords.readthedocs.io/en/latest/