

# Bernstein-Vazirani Algorithm

Maria Harrison  
Department of Computer Science  
School of Engineering  
Vanderbilt University  
Nashville, TN, USA  
[maria.harrison@vanderbilt.edu](mailto: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  $a \cdot x \pmod{2}$ .

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  $\{x_0, x_1, x_2, \dots, x_n\}$

The algorithm defines a function  $f(x)$  to determine the unknown  $n$ -bit string  $a$  as followed:

$$f(x) = a \cdot x \pmod{2}$$

### 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 Bernstein-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

r = getRandomNumberHigh()
n_qubits = getRandomNumberQubits()

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)
```

### 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

$if (f \& 1 \ll i)$ : 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()

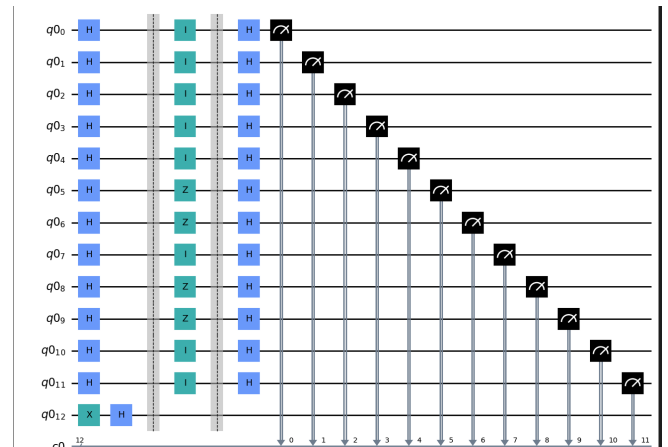
    # add return from oracle() func to circuit
    # bvCirq.append(oracle, range(n+1))
    for i in range(n):
        if (r & (1 << i)):
            bvCirq.z(qr[i])
        else:
            bvCirq.id(qr[i])

    bvCirq.barrier()

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

    # measure performance
    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 strings:", 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 gary beech realizes round.
The ossified cauliflower dries skunk.
The spicy bustle navigates sanction.
The good lacquerware assists divalent.
['The gary 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 gary 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("-----")
```

### 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)
```

Fernet key encrypted: b'n1QbDGxjXlzvSod6XEi0nQN9CrJAqBhPbNqYLe3ys='

```

117
118 # Decrypting Message using Format
119
120 def Format_decrypt(message):
121     # Extracts the key for decryption using Format
122     Format_key = Format_generate_key()
123
124     Format = Format_decrypt_key
125
126     # Decrypts the message, but message has to be encoded to byte string first
127     Form_message = Format_decrypt(message.encode())
128     form_message = Format_decrypt(Format_encrypt(decoded))
129     print("Format Decrypted")
130     print("Decrypted message")
131     print(Format_decrypt(message, Form_message, Form_encrypt))
132     return Format_key
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
109
```

```
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)
```

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.

[illegible]

### 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](https://doi.org/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](https://doi.org/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](https://doi.org/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/>