Middle East Technical University
Department of Computer Engineering
Wireless Systems, Networks and Cybersecurity (WINS) Laboratory

Bellman-Ford Algorithm Implementation on Microsoft Seal-EVA

CENG519 Network Security
2021-2022 Spring
Term Project Report

Prepared by
Berkay Demirören
Student ID: e2309888
berkay.demiroren@metu.edu.tr
Computer Engineering
16 June 2022

Abstract

This report aims to explain what is homomorphic encryption and how Microsoft Seal - EVA will accomplish this task briefly, how Bellman-Ford single-source to all vertexes shortest paths can be implemented on PyEva module and show benchmarks of this implementation with various parameters.

1 Introduction

Homomorphic encryption is a form of encryption that permits users to perform computations on its encrypted data without first decrypting it. This enables server-side applications securely process data. To achieve this Microsoft developed a homomorphic encryption library called SEAL and EVA is the compiler for this domain-specific language. In the scope of this study, a well-known algorithm Bellman-Ford One-Source to All Vertexes Shortest Path is implemented on the module of PyEva which is a python package for EVA that written with C programming language.

With this approach, a user can pass its sensitive data into server as a serialized graph (1D-Array representation of a directed graph) then server which in this study only a function can process that data and receive Bellman-Ford Shortest paths array. Therefore, server will not know what is the sensitive data. This was the aim of this study, however EVA only supports limited set of operations. For instance comparison is not possible which is the main operations for Bellman-Ford algorithm. To achieve that a trusted third party used to compare and feed back into the result.

2 Background and Related Work

2.1 Background

In order to fully understand this report, I will provide background information.

2.1.1 Homomorpyhic Encryption

According to Armknecht et al. in 2015 article A guide to fully homomorphic encryption, Homomorphic encryption is a form of encryption with an additional evaluation capability for computing over encrypted data without access to the secret key. The result of such a computation remains encrypted. Homomorphic encryption can be viewed as an extension of public-key cryptography. Homomorphic refers to homomorphism in algebra: the encryption and decryption functions can be thought of as homomorphisms between plaintext and ciphertext spaces.

Homomorphic encryption includes multiple types of encryption schemes that can perform different classes of computations over encrypted data.[1] The computations are represented as either Boolean or arithmetic circuits. Some common types of homomorphic encryption are partially homomorphic, somewhat homomorphic, leveled fully homomorphic, and fully homomorphic encryption:

Partially homomorphic encryption encompasses schemes that support the evaluation of circuits consisting of only one type of gate, e.g., addition or multiplication. Somewhat homomorphic encryption schemes can evaluate two types of gates, but only for a subset of circuits. Leveled fully homomorphic encryption supports the evaluation of arbitrary circuits composed of multiple types of gates of bounded (pre-determined) depth. Fully homomorphic encryption (FHE) allows the evaluation of arbitrary circuits composed of multiple types of gates of unbounded depth, and is the strongest notion of homomorphic encryption. For the majority of homomorphic encryption schemes, the multiplicative depth of circuits is the main practical limitation in performing computations over encrypted data. Homomorphic encryption schemes are inherently malleable. In terms of malleability, homomorphic encryption schemes have weaker security properties than non-homomorphic schemes.

2.1.2 Microsoft Seal and EVA

Microsoft SEAL is an easy-to-use open-source (MIT licensed) homomorphic encryption library developed by the Cryptography and Privacy Research Group at Microsoft. Microsoft SEAL is written in modern standard C++ and is easy to compile and run in many different environments.

2.1.3 Bellman Ford Algorithm

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.[1] It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively.[2] Edward F. Moore also published a variation of the algorithm in 1959, and for this reason it is also sometimes called the Bellman–Ford–Moore algorithm.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.[3] If a graph contains a ”negative cycle” (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle

2.2 Related Work

The perennial question: Should related work be covered near the beginning of the paper or near the end? **Beginning**: if it can be short yet detailed enough, or if it's critical to take a strong defensive stance about previous work right away. In this case Related Work can be either a subsection at the end of the Introduction, or its own Section 2 [**?**]. **End**: if it can be summarized quickly early on (in the Introduction or Preliminaries), or if sufficient comparisons require the technical content of the paper. In this case Related Work should appear just before the Conclusions, possibly in a more general section ‘Discussion and Related Work’’ [**?**].

3 Main Contributions

3.1 Bellman-Ford Implementation

v Above psudo-code is the steps for implementing Bellman-Ford Algorithm. My implementation also builds on top of that.

```
def BellmanFord(self, src):

    # Step 1: Initialize distances from src to all other vertices
    # as INFINITE
    dist = [float("Inf")] * self.V
    dist[src] = 0


    # Step 2: Relax all edges |V| - 1 times. A simple shortest
    # path from src to any other vertex can have at-most |V| - 1
    # edges
    for _ in range(self.V - 1):
```

```python
        # Update dist value and parent index of the adjacent vertices of
        # the picked vertex. Consider only those vertices which are still in
        # queue
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

    # Step 3: check for negative-weight cycles. The above step
    # guarantees shortest distances if graph doesn't contain
    # negative weight cycle. If we get a shorter path, then there
    # is a cycle.

    for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    print("Graph contains negative weight cycle")
                    return

    # print all distance
    self.printArr(dist)
```

Above code snippet is the most basic python implementation of Bellman-Ford Algorithm. While my implementation uses that it also includes extra configurations and functions. First of all, EVA's set of instructions are very limited. Most of the basic operands are missing, such as; comparison, sorting. I added and explained my implementation below.

```python
class Graph:

  def __init__(self, num_of_vertices):
    self._num_of_vertices = num_of_vertices # No. of vertices
    self._graph = []

  # function to add an edge to graph
  def addEdge(self, u, v, w):
    self._graph.append([u, v, w])

  # utility function used to print the solution
  def printArr(self, dist):
    print("Vertex Distance from Source")
    for i in range(self._num_of_vertices):
      print("{0}\t\t{1}".format(i, dist[i]))

  def printGraph(self):
    for cell in self._graph:
      u, v, w = cell[0],cell[1],cell[2]
      print(f'{u}-{v} : {w}')

  def generate_random_graph(self, p):
    for u in range(0,self._num_of_vertices):
      for v in range(0, self._num_of_vertices):
        w = randint(-p,p)
        if abs(w) < 4:
          w = 0
        self.addEdge(u, v, w)
```

Above Graph class is for the setting of implementation. It consists of init, addEdge, generateRandomGraph methods as building blocks. Also printArr and printGraph for its representation purposes.

```python
def serializeGraphZeroOne(GG,vec_size):
    n = GG.size()
    graphdict = {}
    g = []
    for row in range(n):
        for column in range(n):
            if GG.has_edge(row, column) or row==column: # I assumed the vertices are connected to themselves
                weight = 1
            else:
                weight = 0
            g.append( weight  )
            key = str(row)+'-'+str(column)
            graphdict[key] = [weight] # EVA requires str:listoffloat
    # EVA vector size has to be large, if the vector representation of the graph is smaller, fill the eva vector with zeros
    for i in range(vec_size - n*n):
        g.append(0.0)
    return g, graphdict

def prepareInput(n, m):
  global GG
  GG = Graph(n)
  GG.generate_random_graph(10)
  input = {}
  gg = generateGraph(n,3,0.5)
  graph, graphdict = serializeGraphZeroOne(gg,m)
  input['Graph'] = graph
  return input
```

Serialize and prepare input functions are directly related with EVA programming. Since EVA's driver program only works with well-structured special types of inputs blocks. Which means,

```

class EvaProgramDriver(EvaProgram):
  def __init__(self, name, vec_size=4096, n=4):
    self.n = n
    super().__init__(name, vec_size)

  def __enter__(self):
    super().__enter__()

  def __exit__(self, exc_type, exc_value, traceback):
    super().__exit__(exc_type, exc_value, traceback)
```

EvaProgramDriver class used to run EvaPrograms with given configurations.

```
def graphanalticprogram(graph):
  global GG
  vertice_count = GG._num_of_vertices
  dist = [float("Inf")] * vertice_count
  dist[src] = 0


  for _ in range(vertice_count - 1):
    for u, v, w in GG._graph:
      if dist[u] != float("Inf") and dist[u] + w < dist[v]:
          dist[v] = dist[u] + w

  for u, v, w in GG._graph:
      if dist[u] != float("Inf") and dist[u] + w < dist[v]:
          print("Graph contains negative weight cycle")
          return graph<<0
  self.printArr(dist)
  return graph<<1
```

This part is where main algorithm run.

```python
def simulate(n):
  m = 4096
  print("Will start simulation for ", n)
  config = {}
  config['warn_vec_size'] = 'false'
  config['lazy_relinearize'] = 'true'
  config['rescaler'] = 'always'
  config['balance_reductions'] = 'true'
  inputs = prepareInput(n, m)

  graphanaltic = EvaProgramDriver("graphanaltic", vec_size=m,n=n)
  with graphanaltic:
    graph = Input('Graph')
    reval = graphanalticprogram(graph)
    Output('ReturnedValue', reval)

  prog = graphanaltic
  prog.set_output_ranges(30)
  prog.set_input_scales(30)

  start = timeit.default_timer()
  compiler = CKKSCompiler(config=config)
  compiled_multfunc, params, signature = compiler.compile(prog)
  compiletime = (timeit.default_timer() - start) * 1000.0 #ms

  start = timeit.default_timer()
  public_ctx, secret_ctx = generate_keys(params)
  keygenerationtime = (timeit.default_timer() - start) * 1000.0 #ms

  start = timeit.default_timer()
  encInputs = public_ctx.encrypt(inputs, signature)
  encryptiontime = (timeit.default_timer() - start) * 1000.0 #ms

  start = timeit.default_timer()
  encOutputs = public_ctx.execute(compiled_multfunc, encInputs)
  executiontime = (timeit.default_timer() - start) * 1000.0 #ms

  start = timeit.default_timer()
  outputs = secret_ctx.decrypt(encOutputs, signature)
  decryptiontime = (timeit.default_timer() - start) * 1000.0 #ms

  start = timeit.default_timer()
  reference = evaluate(compiled_multfunc, inputs)
  referenceexecutiontime = (timeit.default_timer() - start) * 1000.0 #ms

  # Change this if you want to output something or comment out the two lines below
  for key in outputs:
    print(key, float(outputs[key][0]), float(reference[key][0]))

  mse = valuation_mse(outputs, reference) # since CKKS does approximate computations,
     this is an important measure that depicts the amount of error

  return compiletime, keygenerationtime, encryptiontime, executiontime,
    decryptiontime, referenceexecutiontime, mse
```

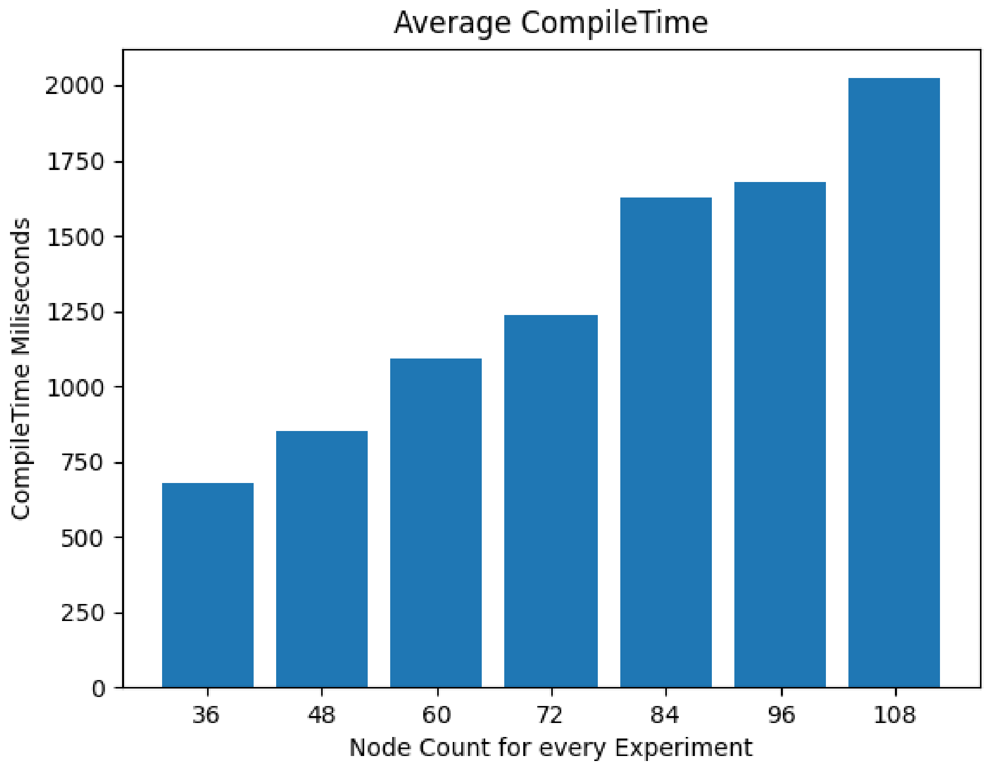
Simulation makes possible to benchmark code with different parameters.

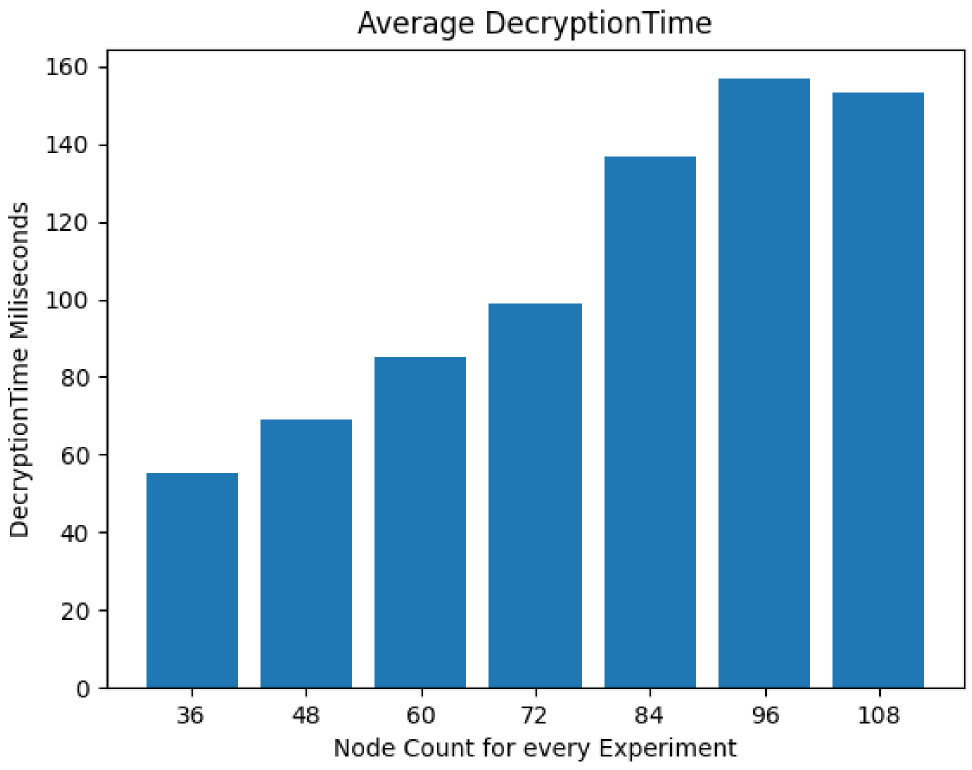
```python

if __name__ == "__main__":
  simcnt = 100 #The number of simulation runs, set it to 3 during development
    otherwise you will wait for a long time
  # For benchmarking you must set it to a large number, e.g., 100
  #Note that file is opened in append mode, previous results will be kept in the file
  resultfile = open("results.csv", "a")  # Measurement results are collated in this
    file for you to plot later on
```

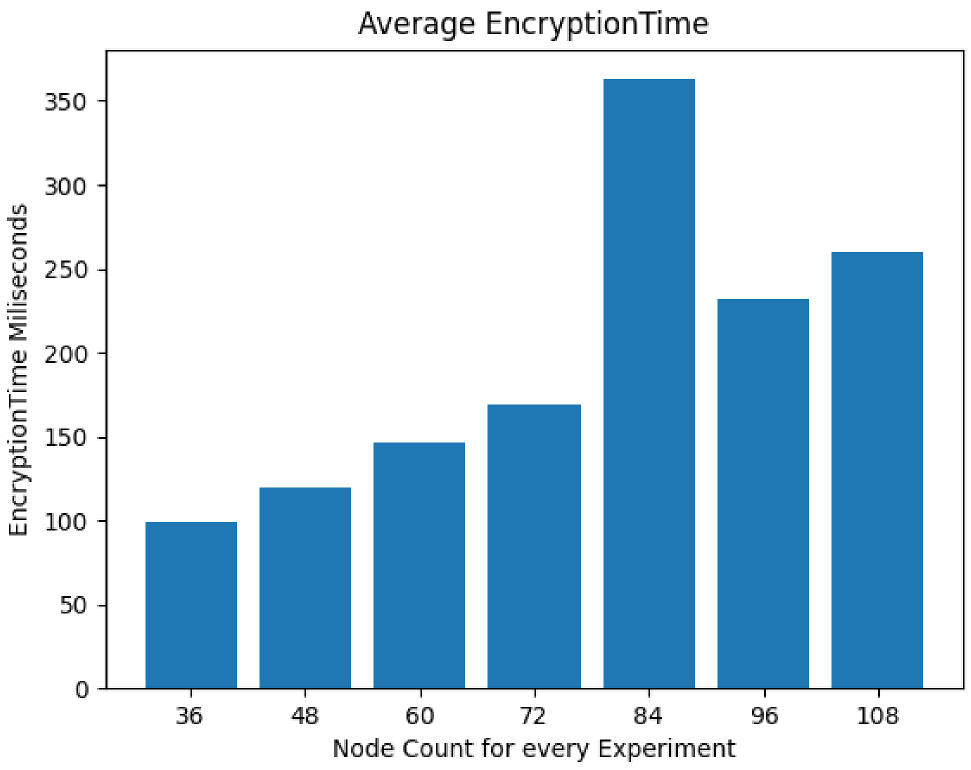
```python
resultfile.write("NodeCount,PathLength,SimCnt,CompileTime,KeyGenerationTime,
  EncryptionTime,ExecutionTime,DecryptionTime,ReferenceExecutionTime,Mse\n")
resultfile.close()

print("Simulation campaing started:")
for nc in range(36,64,4): # Node counts for experimenting various graph sizes
  n = nc
  resultfile = open("results.csv", "a")
  for i in range(simcnt):
    #Call the simulator
    compiletime, keygenerationtime, encryptiontime, executiontime, decryptiontime,
  referenceexecutiontime, mse = simulate(n)
    res = str(n) + "," + str(i) + "," + str(compiletime) + "," + str(
  keygenerationtime) + "," +  str(encryptiontime) + "," +  str(executiontime) + ","
   +  str(decryptiontime) + "," +  str(referenceexecutiontime) + "," +  str(mse) +
  "\n"
    print(res)
    resultfile.write(res)
  resultfile.close()

```

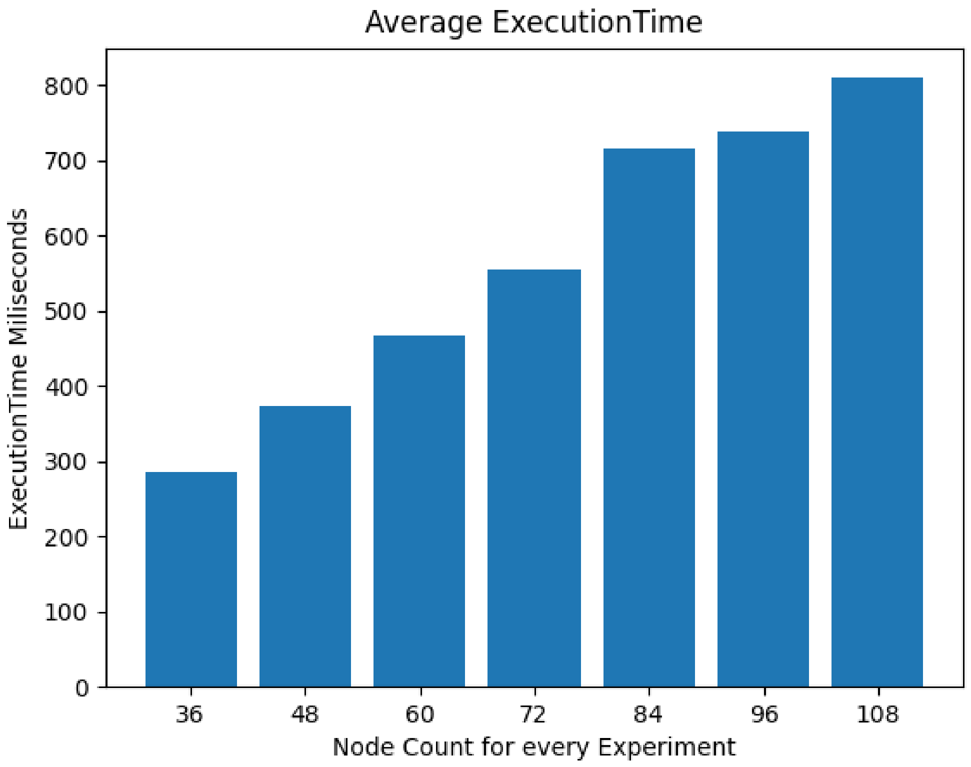
It is the driver function for simulation. That saves metrics that measured in the simulation into a csv file.

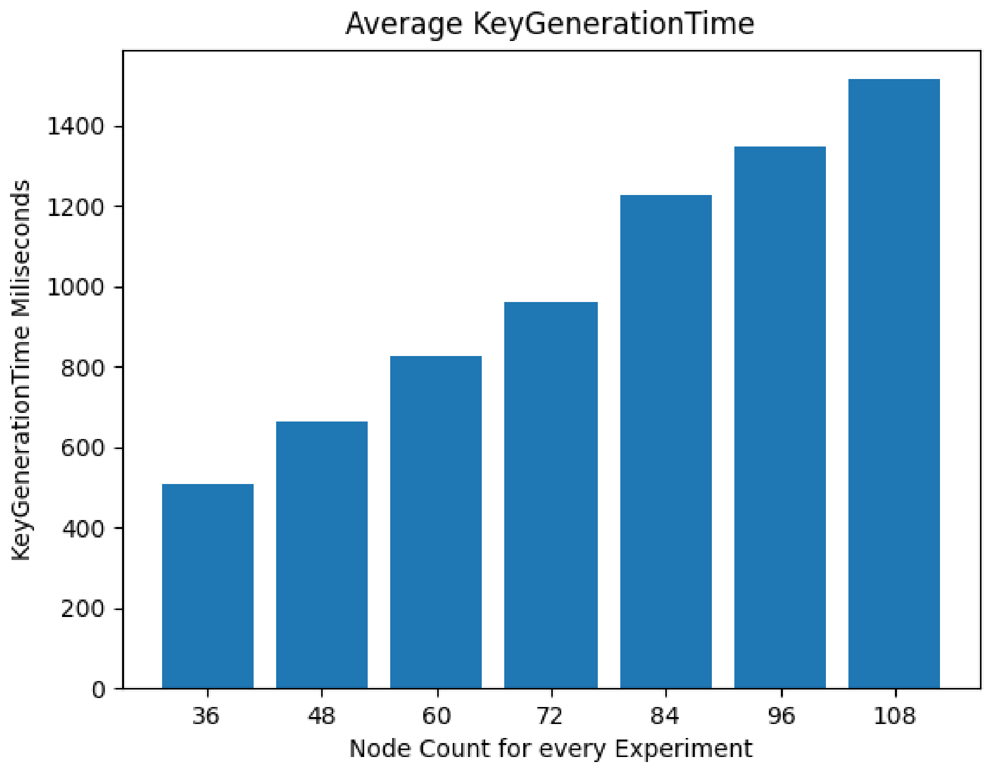
4 Results and Discussion

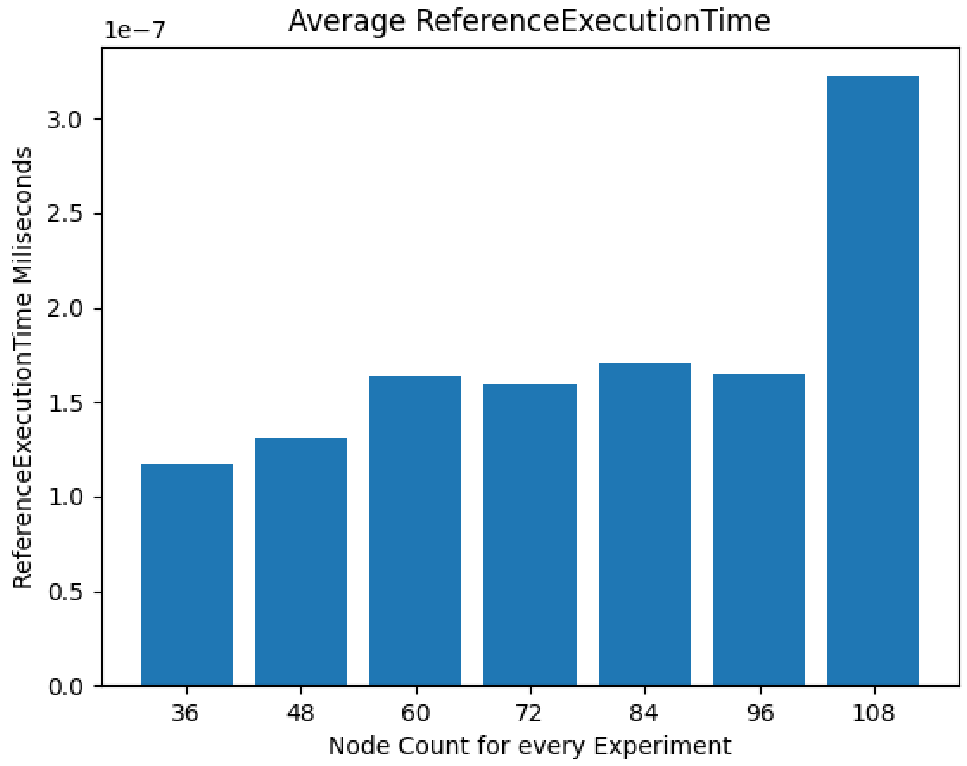





Average EncryptionTime
EncryptionTime Miliseconds
0
50
100
150
200
250
300
350
36
48
60
72
84
96
108
Node Count for every Experiment


Average ExecutionTime
ExecutionTime Miliseconds
0
100
200
300
400
500
600
700
800
36
48
60
72
84
96
108
Node Count for every Experiment





Above graphs show that while node count increases also time to process data increases as expected