CENG519 Network Security

Bellman-Ford Algorithm Implementation on Microsoft Seal-EVA

Berkay Demirören

Wireless Systems, Networks and Cybersecurity Laboratory
Department of Computer Engineering
Middle East Technical University
Ankara Turkey

Outline of the Presentation

Agenda

Introduction

- Homomorphic Encryption
- Micrsoft Seal - EVA
- Bellman-Ford Algorithm

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.

Agenda

Homomorphic 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.

Microsoft Seal - 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.

Bellman-Ford Algorithm

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source to all of the other vertices in a weighted digraph.It is capable of handling graphs in which some of the edge weights are negative numbers. Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.

If a graph contains a ”negative cycle” that is reachable from the source, then there is no cheapest path. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle

Agenda

Implementation

Algorithm 1: Bellman Ford Algorithm

$\forall v \in V, d[v] \leftarrow \infty$ // set initial distance estimates
//optional: set $\pi(v) \leftarrow$nil for all v, $\pi(v)$ represents the predecessor of v
$d[s] \leftarrow 0$ // set distance to start node trivially as 0
for i *from* $1 \rightarrow n-1$ **do**
  for $(u, v) \in E$ **do**
    $d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$ // update estimate of v
    // optional - if $d[v]$ changes, then $\pi(v) \leftarrow u$
// Negative Cycle Step
for $(u, v) \in E$ **do**
  if $d[v] > d[u] + w(u, v)$ **then**
    return "Negative Cycle"; // negative cycle detected
return $d[v] \; \forall \; v \in V$

Implementation

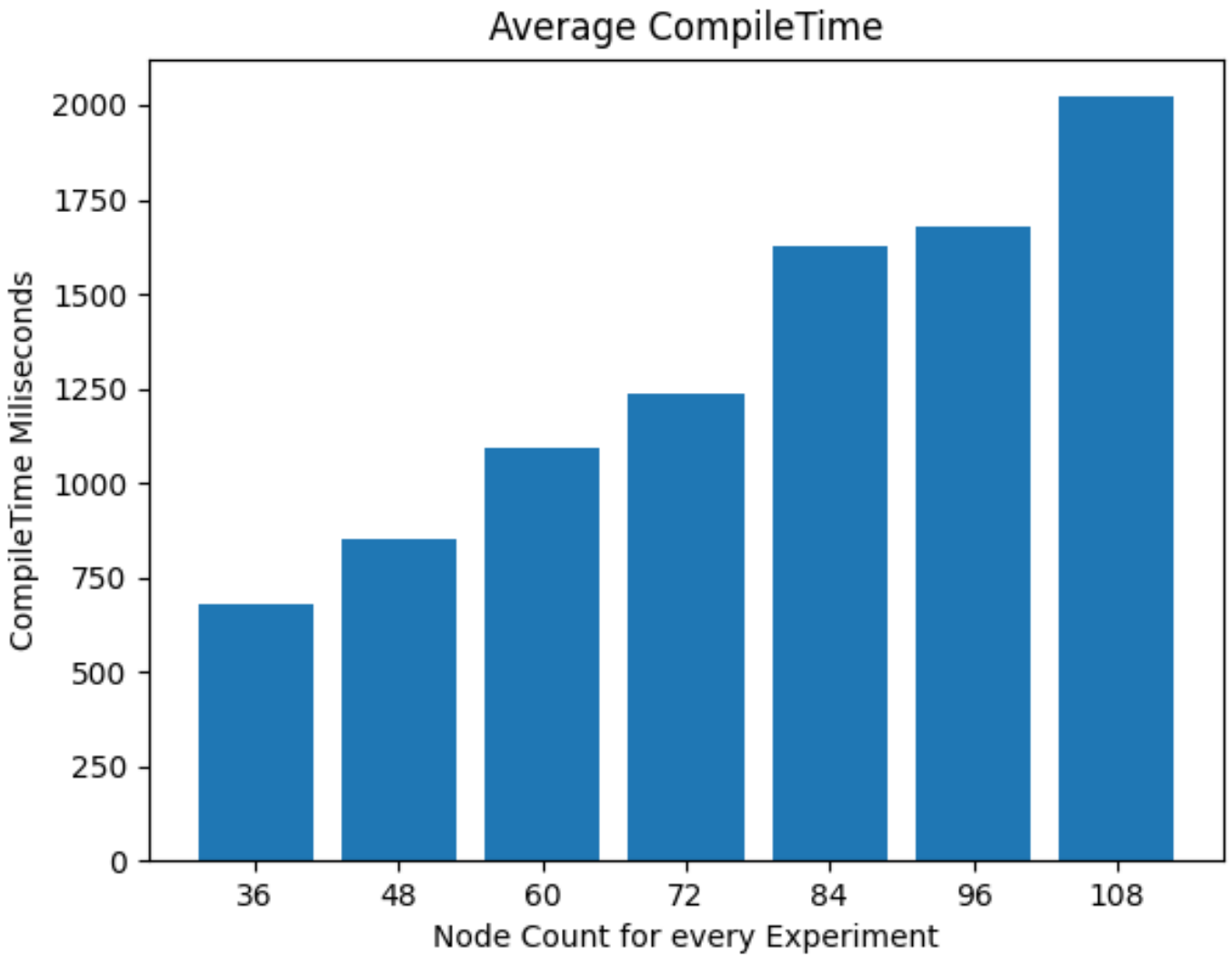
```
class Graph:
    def __init__(self, num_of_vertices):
        self._num_of_vertices = num_of_vertices # No. of vertices
        self._graph = []

    def addEdge(self, u, v, w):
        self._graph.append([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)
```

Implementation

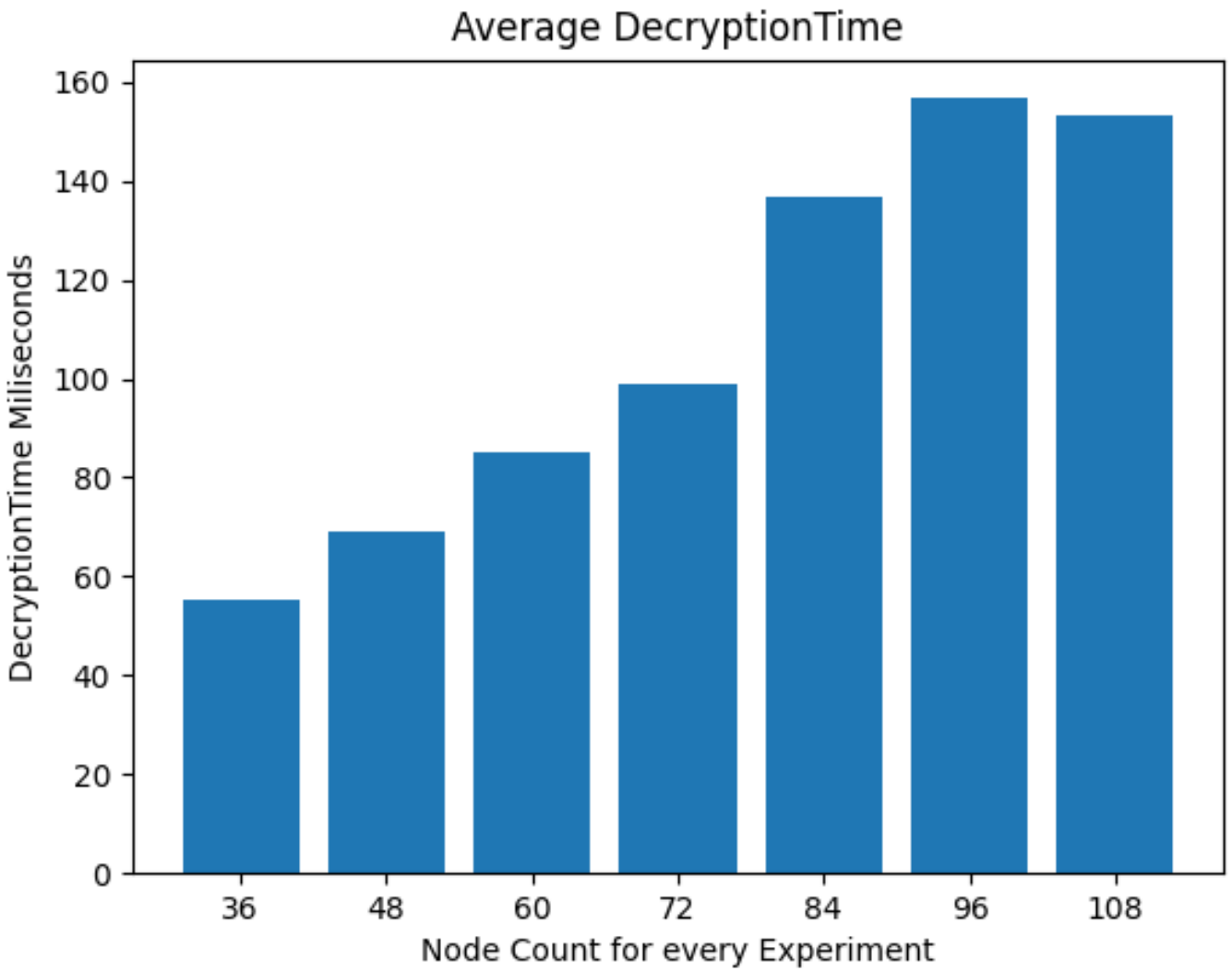
```
def BellmanFord(self, src):

        dist = [float("Inf")] * self.V
        dist[src] = 0

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

        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

        self.printArr(dist)
```

Agenda

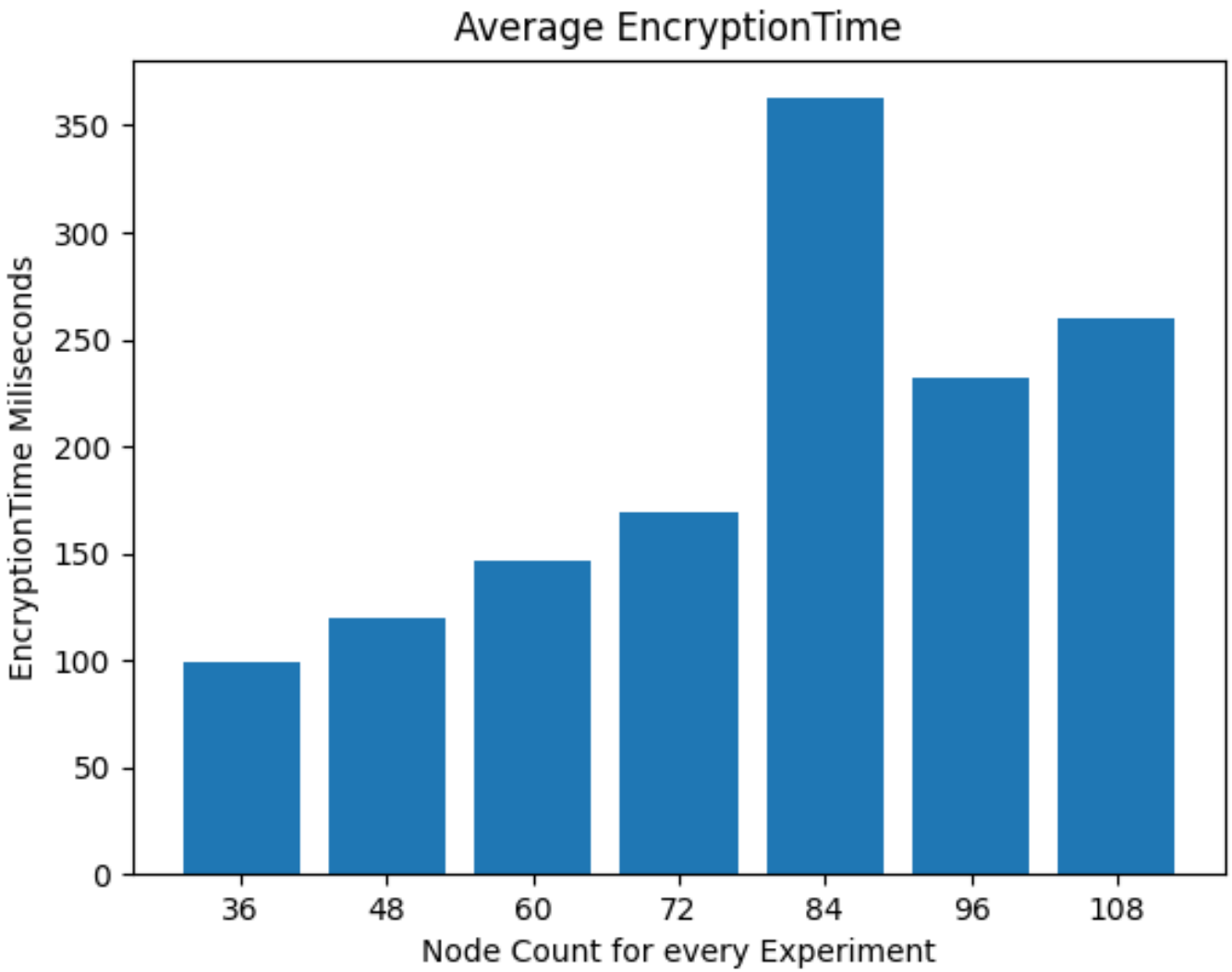
Results


Average CompileTime
CompileTime Miliseconds
2000
1750
1500
1250
1000
750
500
250
0
36
48
60
72
84
96
108
Node Count for every Experiment

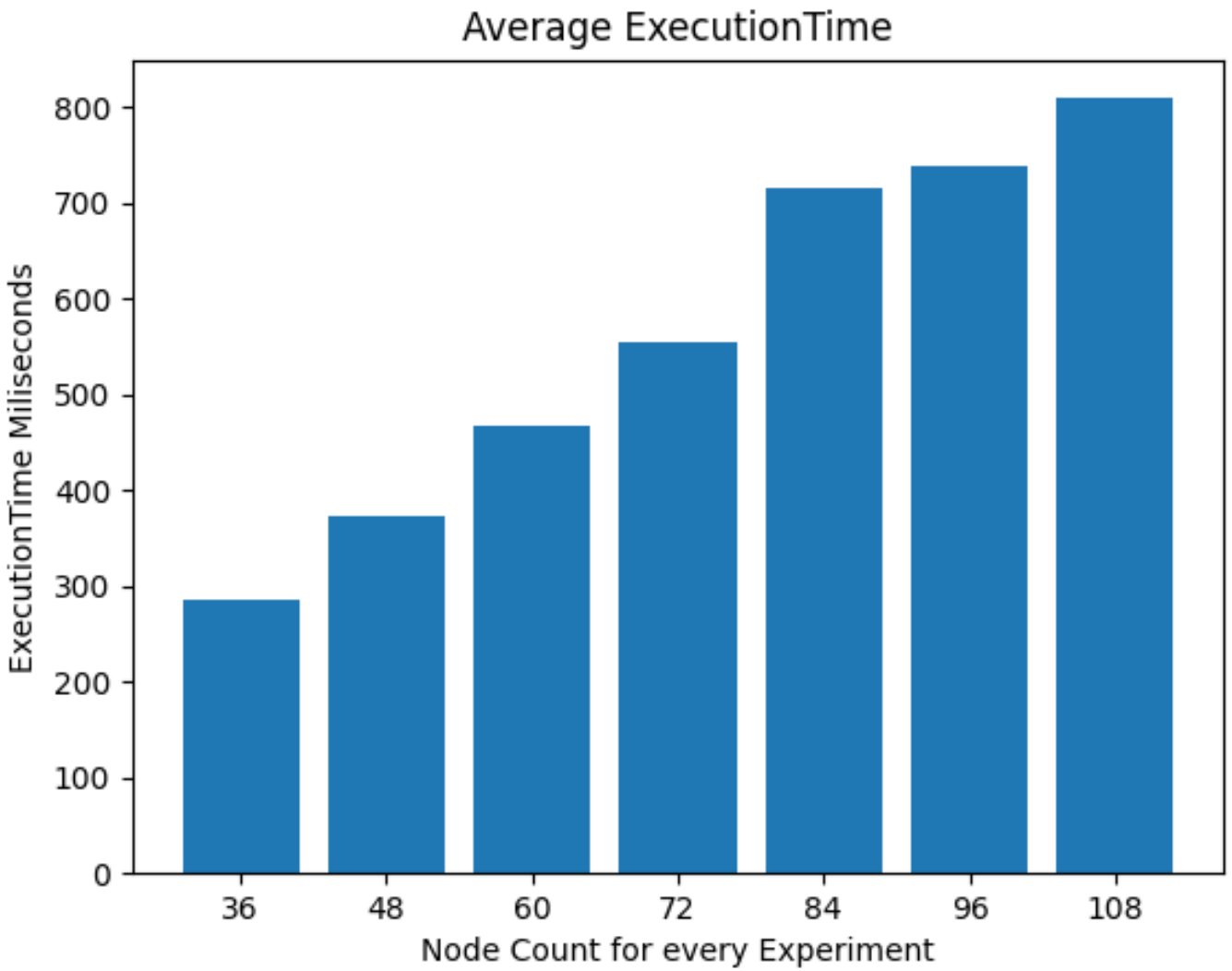
Results


Average DecryptionTime
DecryptionTime Miliseconds
0
20
40
60
80
100
120
140
160
36
48
60
72
84
96
108
Node Count for every Experiment

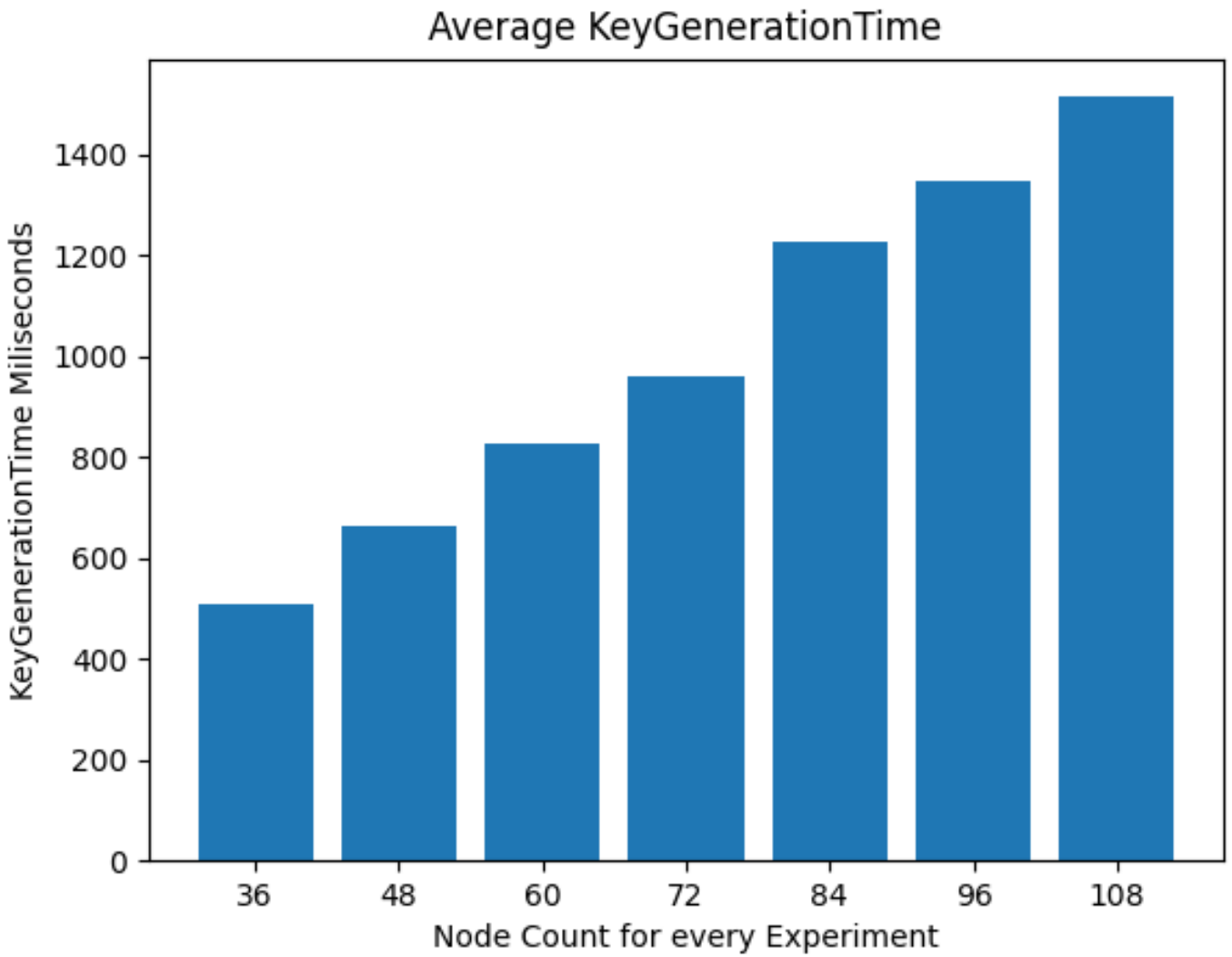
Results


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

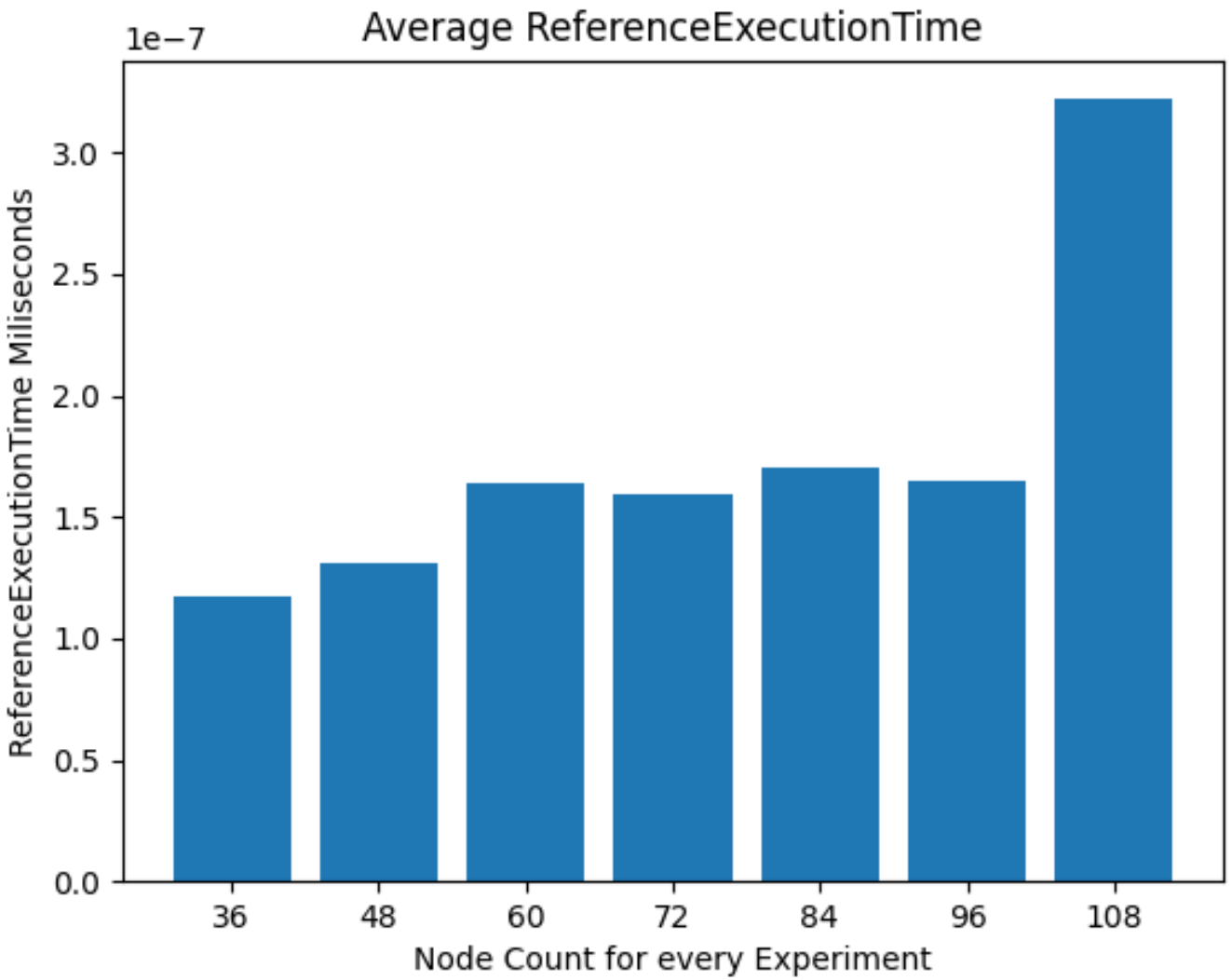
Results


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

Results


Average KeyGenerationTime
KeyGenerationTime Miliseconds
0
200
400
600
800
1000
1200
1400
36
48
60
72
84
96
108
Node Count for every Experiment

Results


Average ReferenceExecutionTime
1e−7
ReferenceExecutionTime Miliseconds
3.0
2.5
2.0
1.5
1.0
0.5
0.0
36
48
60
72
84
96
108
Node Count for every Experiment

Agenda

Conclusions

- Homomorphic Encryption
- Micrsoft Seal - EVA
- Bellman-Ford Algorithm
- Benchmark Resu

Questions

THANK YOU

CENG519 Network Security

Bellman-Ford Algorithm Implementation on Microsoft Seal-EVA

presented by Berkay Demirören