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

Secure Implementation of Topological Sort of a Directed Acyclic Graph Algorithm using Homomorphic Encryption

CENG519 Network Security
2021-2022 Spring
Term Project Report

Prepared by
Berkan Sözen
Student ID: 2594927
berkan.sozen@metu.edu.tr
Computer Engineering
3 June 2022

Abstract

In this work, a secure implementation of topological sort of a DAG (Directed Acyclic Graph) is implemented between a server and client using homomorphic encryption. The system is built on a mock client-server architecture where client encrypts a list representation of a DAG using Microsoft SEAL library, and server makes computational operations on encrypted data. The operations that can run on encrypted data are limited, hence the more complicated operations need to be dissembled to more basic operations, making the implementation challenging. The operations that can neither be simplified nor available are instead sent back to client, which returns the outcome to server. This results in a slow solution; but ensures that the data is not decrypted at the server.

Table of Contents

List of Figures

1 Introduction

Encryption of data is usually relevant for storing or transportation stages of data. This is partially because traditional cryptography techniques require the data to be unencrypted to be able to process it correctly, which implies that at some stage on a supposedly secure system, the data has to be decrypted and processed in plaintext. Examples to this situation are; sensitive information of companies being processed by 3rd parties, or data analytics extracted from personal data, and other situations where the data is entrusted with the hope that is won't be misused or get stolen [2].

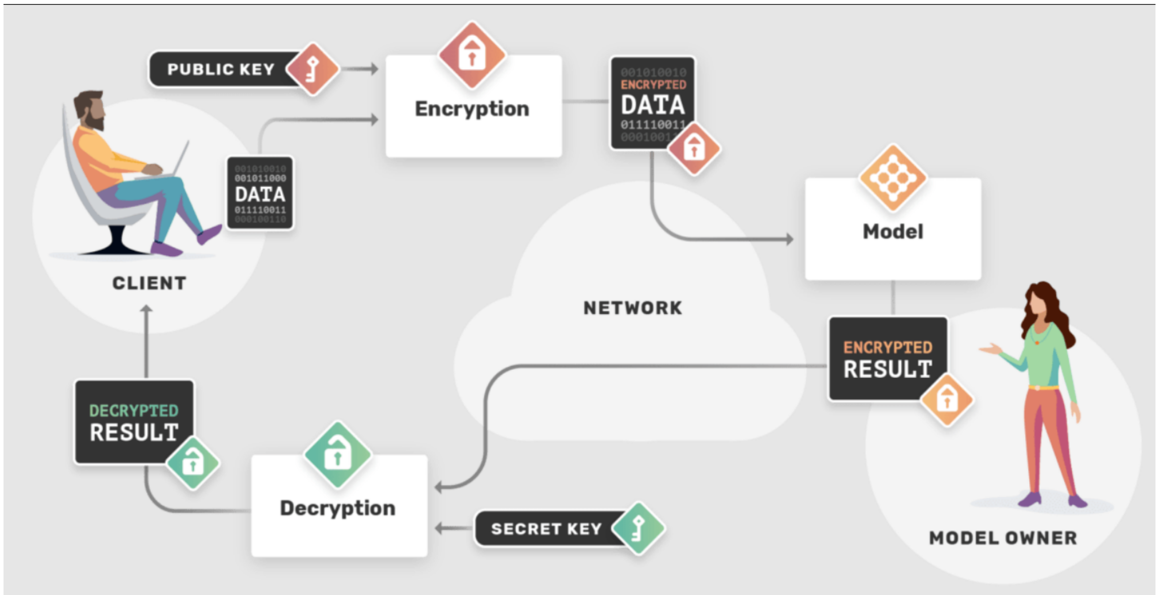

Figure 1 . Homomorphic Encryption [1]

A viable solution to this problem Homomorphic encryption. In principle, the term "homomorphic encryption" generalises encryption techniques that "allow encrypted data to be processed as if it were in plaintext and produce the correct value once decrypted" [3]. However, in order to keep the data safe, the operations that server can run on encrypted data are need to be limited. Currently there are three types of homomorphic encryption scheme that is categorized by the operations they provide [2]:

1. Partially homomorphic encryption: only one mathematical operation can be performed an unlimited number of times on the encrypted data.
2. Somewhat homomorphic encryption: any number of additions but only one multiplication operation can be performed on the encrypted data.
3. Fully homomorphic encryption: any number of additions and multiplications operation can be performed on the encrypted data.

While these limitations help prevent unauthorized decryption, implementing the algorithms that will process the data becomes challenging. The technique demonstrated in this project is fully homomorphic encryption. A major data representation and processing model makes use of Graph Theory; therefore we implement Topological Sorting on Directed Acyclic Graphs using fully homomorphic encryption. The encryption algorithm is provided by Microsoft

SEAL library [4].

2 Background and Related Work

2.1 Background

The project requires prerequisites on graph theory, programming with Python and general knowledge of server-client architecture.

2.2 Related Work

While homomorphic encryption is not a new technique, it is still not very wide-spread, resulting in scarce related work.

Some libraries that are currently available are:

1. A fully homomorphic encryption library is PALISADE [5]
2. awesome-he by jonaschn [6]
3. fully-homomorphic-encryption by Google [7]
4. TFHE: Fast Fully Homomorphic Encryption over the Torus [8]

Microsoft SEAL is considered "leveled fully homomorphic" by some sources [9]

3 Main Contributions

3.1 Implementation of Comparison

The algorithm to be implemented is a graph sorting algorithm, and it requires comparisons on data in regular domain. However fully homomorphic encryption provides limited operations: the Microsoft SEAL library which is the backbone of the encryption system does not allow comparison operations. This problem is solved by introducing a mechanism between the server and the client to provide comparison operation.

Since the comparison operation is not an option in the encrypted data on the server side, an interface with the client is implemented. This interface should allow client to be invoked from the server, and it should allow the server to continue the processing from where it left off. In other words, the comparison operation is temporarily handed back to client to make up for the limited availability of mathematical operations on server side.

3.2 Client-Server Interface

The need for such an interface is required because the server can not handle the comparison. This interface should satisfy two fundamental flows besides comparison implementation:

1. Server should invoke the client whenever necessary and the client should be able to handle this interrupt.
2. After client is processed the data and invoke the server, server should run from where it left of.

Also the data to be compare should be managed in the server. That means instead of sending data, use multiplication with an array that includes only one 1 and the other items are zero.

That results with an array that either all zero or has only one item equals to 1. Therefore index can be detected but data itself can not be accessed.

4 Methodology

The code simulates the selected algorithm securely. When it runs, it automatically creates a DAG with selected node size and makes operations on this graph. After resulting with the simulation, it generates results file, plots figures and, draws a visual representation of the graph. The code can be seen in Appendix C.

4.1 Server Implementation

The server -which is represented by a function- should implement the selected algorithm: Kahn's Algorithm for Topological Sorting [10]. As stated on previous sections the limitations should be handled by the server by implementing client-server interface. To satisfy the requirement of client-server interface, following features should be added to server code:

1. Server should hold a state. The state is also the main communication mechanism between client and the server.
2. Server should yield a state-result pair instead of returning. Yielding provides both invoking the client and the resuming from yield location.

Server is implemented in such a manner that the only encrypted data is the graph itself. The helper variables such as in-degree array and state of the process is held in global scope. Therefore they are not protected. The reason behind that is keeping the encryption operations simple as possible due to scope of the assignment. But all of the operations can also be applied the helper variables.

There are two types of states that server provides:

1. State "check if all zero": It is yielded with an encrypted array that has either all of its items are zero or only the first item is one
2. State "finished": Sent when the sorting operation is completed.

When the only incoming states which is "not all zero" is set by the client, server knows comparison returns true.

4.1.1 Kahn's Algorithm for Topological Sorting

Once all the required mathematical operations are provided either by the encryption library or via client-server communication; Kahn's algorithm for topological sorting can be implemented by executing the following steps [10]:

1. Compute in-degree (number of incoming edges) for each of the node.
2. Pick all the nodes with in-degree as 0 and add them into a queue: This operation requires comparing in-degree items with 0.
3. Pop a node from the queue.
4. Decrease in-degree by 1 for all its neighbouring nodes. If in-degree of a neighbouring node is reduced to zero, then add it to the queue: This operations requires finding neighbours which includes comparison operation.
5. Repeat Step 3 until the queue is empty.

4.2 Client Implementation

The client is responsible for providing graph and necessary helper variables stated in the server implementation section. Also it should handle the basic requests from the server by using client-server interface. To satisfy the requirement of client-server interface, following features should be added to client code:

1. Since the server yields results, the client should iterate the server by calling next on the generator function of the server.
2. For every iteration, client should check the state in the response from server and act according to it.

The incoming states is handled as follows:

1. State "check if all zero": This state means that server needs a comparison operation. The resulting array will be decrypted by the client and content will be checked if all the items are zero or not. If the all items are not zero, client will set global state to "not all zero" and iterate the server.
2. State "finished": This state means sorting operation is completed and the result is achieved.

4.3 Development Environment

The building and running environment is Ubuntu Linux that runs on a Docker container. The related Dockerfile can be seen in Appendix A. Microsoft SEAL encryption library used as the core of the homomorphic encryption. The mock server-client architecture is implemented using Python programming language on Visual Studio Code editor.

5 Results and Discussion

5.1 Results

As we can see in the selected graph in Figure 2, there is a linear relationship between resources consumed and the node count. The pitfalls represent that algorithm is failed to find a topological sort, therefore run time is decreased at that points. Related graphs can be seen in the Appendix A.

5.2 Discussion

The results can be observed from two different perspectives. First of them is the correctness of the results and the other is efficiency of the algorithm. Correctness of the algorithm is verified by network library functions and it is seen that results are not correct sometimes. The pitfall in the graphs shows that algorithm is failed and therefore run time is slow. Efficiency of the algorithm, on the other hand, should be increased. Because the data is transmitted between server and the client multiple times for only one operation, a huge overhead is included in the system. The graph that shows total time required by the node count below shows that.

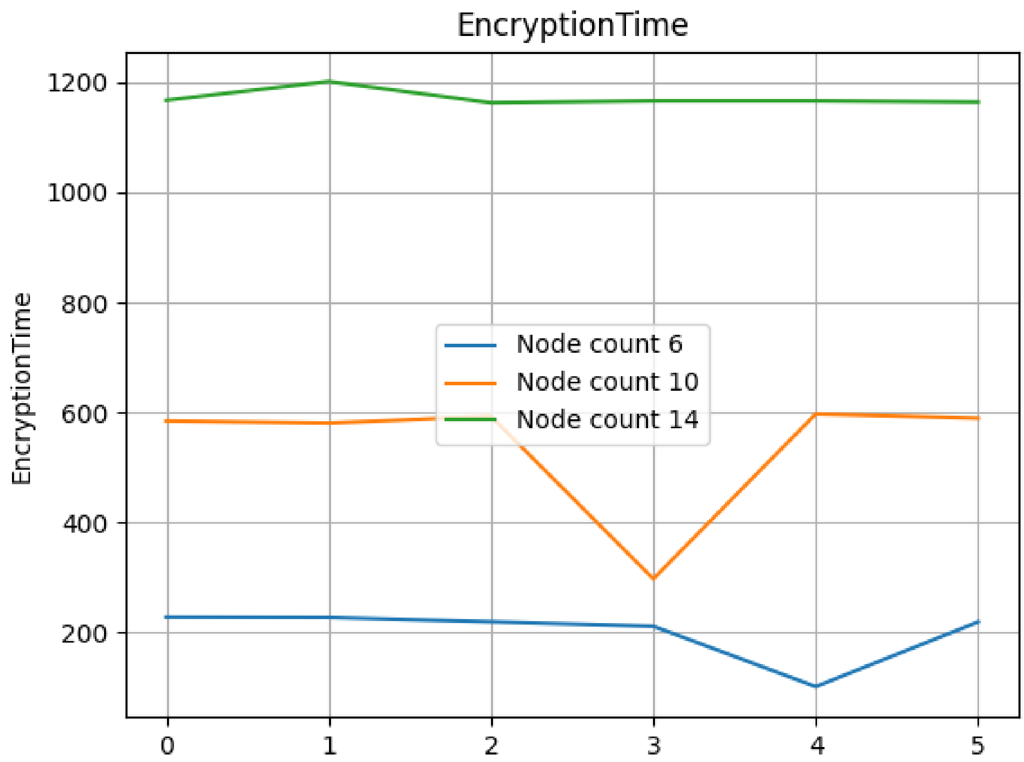

Figure 2 . Execution Time

6 Conclusion

The project is developed as a proof-of-concept work. That means only the graph data itself is encrypted, not the other variables. Therefore there are several security vulnerabilities. The helper variables that are not encrypted can be observed and the data becomes a subject to reverse engineering. Also the global state can be tracked to observe the current state of server. The final output is not also encrypted because of simplicity. To overcome such an important issue all of the variables can be encrypted alongside with the graph and necessary functions can be implemented in the client-server interface if can not be implemented in the server. Also such algorithms can be designed to eliminate the extra data transfer between server and the client.

References

[1] Homomorphic encryption. [Online]. Available: https://research.aimultiple.com/wp-content/webp-express/webp-images/uploads/2021/08/homomorphic-enc-1160x603.png.webp

[2] R. Yackel. (2021) What is homomorphic encryption, and why isn't it mainstream? [Online]. Available: https://www.keyfactor.com/blog/what-is-homomorphic-encryption/

[3] R. K. K. Mark A. Will. (2022) A guide to homomorphic encryption. [Online]. Available: https://learning.oreilly.com/library/view/the-cloud-security/9780128017807/B9780128015957000057.xhtml#s0050

[4] (2019) Microsoft seal. [Online]. Available: https://web.archive.org/web/20190530060553/https://www.microsoft.com/en-us/research/project/microsoft-seal/

[5] Palisade homomorphic encryption software library. [Online]. Available: https://palisade-crypto.org/

[6] Awesome homomorphic encryption. [Online]. Available: https://github.com/jonaschn/awesome-he

[7] Fully homomorphic encryption (fhe). [Online]. Available: https://github.com/google/fully-homomorphic-encryption

[8] Tfhe: Fast fully homomorphic encryption over the torus. [Online]. Available: https://github.com/google/fully-homomorphic-encryption

[9] (2022) Homomorphic encryption. [Online]. Available: https://en.wikipedia.org/wiki/Homomorphic_encryption#Fourth-generation_FHE

[10] (2022) Kahn's algorithm for topological sorting. [Online]. Available: https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/

Appendix A Result Graphs

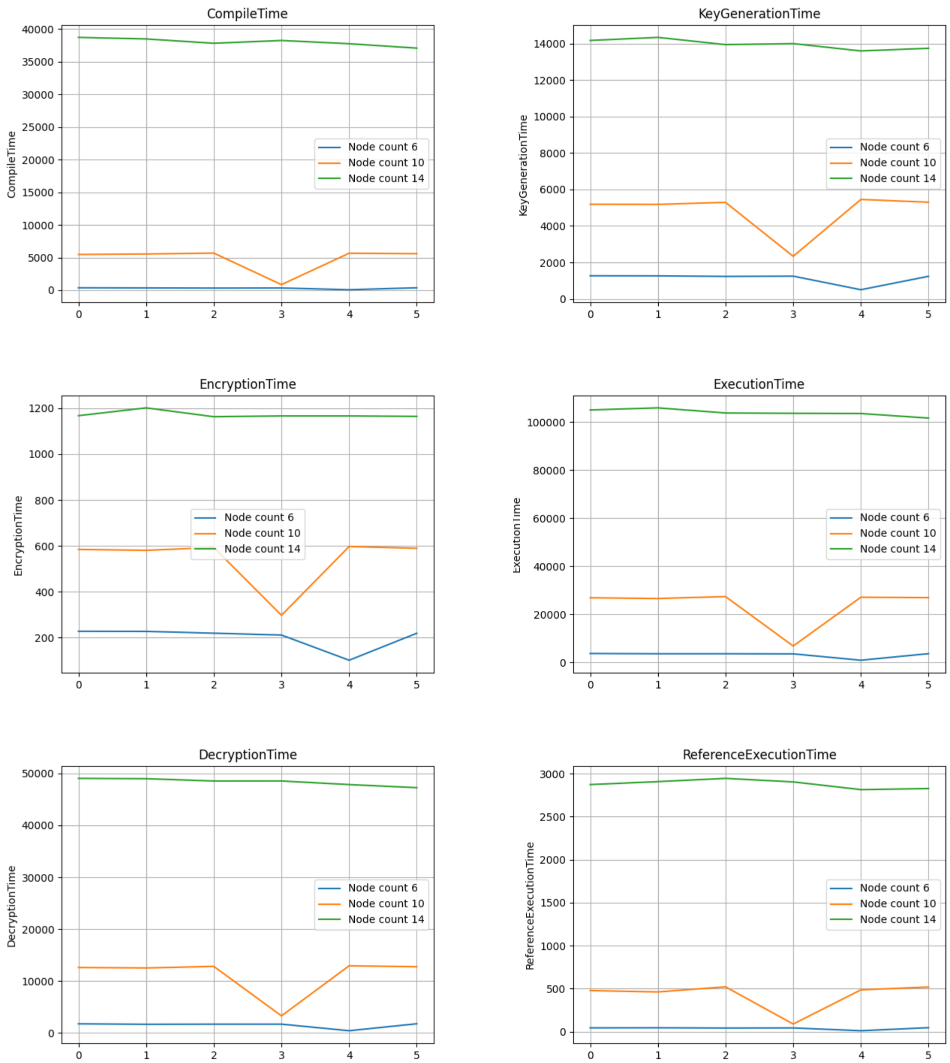

Figure 3 . Results

Appendix B Docker File

```
#Download base image ubuntu 20.04
FROM ubuntu:20.04
# LABEL about the custom image
LABEL maintainer="ertanon@gmail.com"
LABEL version="0.1"
LABEL description="This is a Microsoft EVA development environment image based on
    Ubuntu 20.04"

# Disable Prompt During Packages Installation
ARG DEBIAN_FRONTEND=noninteractive



RUN apt update \
    && mkdir development \
    && cd development \
    && apt install -y python3 python3-dev python3-pip  \
    && cd /usr/local/bin \
    && ln -s /usr/bin/python3 python \
    && pip3 --no-cache-dir install --upgrade pip \
    && apt-get install -y --no-install-recommends apt-utils build-essential sudo git
    cmake \
    && apt install -y libboost-all-dev libprotobuf-dev protobuf-compiler \
    && apt install -y clang  \
    && update-alternatives --install /usr/bin/cc cc /usr/bin/clang 100  \
    && update-alternatives --install /usr/bin/c++ c++ /usr/bin/clang++ 100 \
    && pip3 install numpy

WORKDIR /development

RUN git clone -b v3.6.4 https://github.com/microsoft/SEAL.git \
    && cd SEAL \
    && cmake -S . -B build -DSEAL_THROW_ON_TRANSPARENT_CIPHERTEXT=ON \
    && cmake --build build \
    && cmake --install build

RUN git clone https://github.com/microsoft/EVA.git \
    && cd EVA \
    && git submodule update --init \
    && cmake . \
    && make -j \
    && pip3 install -e ./python \
    && pip3 install -r examples/requirements.txt \
    && cd examples/

RUN pip3 install adhoccomputing \
    && pip3 install networkx

COPY 519ProjectTemplate /development/519ProjectTemplate
```

Appendix C Server-Client Code

```
from itertools import tee
from termios import tcdrain
from eva import EvaProgram, Input, Output, evaluate
from eva.ckks import CKKSCompiler
from eva.seal import generate_keys
from eva.metric import valuation_mse
import timeit
import networkx as nx
from random import random
from os import mkdir
import copy

# Helper packages
import matplotlib.pyplot as plt

# Using networkx, generate a random directed acyclic graph
def generateGraph(n, prob):
    G = nx.gnp_random_graph(n, prob, directed=True)
    nodes = [(u,v) for (u,v) in G.edges() if u<v]
    DAG = nx.DiGraph(nodes)

    plt.tight_layout()
    nx.draw_networkx(DAG, arrows=True)
    plt.savefig(f"results/DAG_{n}.png", format="PNG")
    plt.clf()

    return DAG, nodes

# If there is an edge between two vertices its weight is 1 otherwise it is zero
# Two dimensional adjacency matrix is represented as a vector
# Assume there are n vertices
# (i,j)th element of the adjacency matrix corresponds to (i*n + j)th element in the
    vector representations
def serializeGraphZeroOne(GG,vec_size):
    n = len(GG.nodes())
    graphdict = {}
    g = []
    for row in range(n):
        for column in range(n):
            if GG.has_edge(row, column):
                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

# To display the generated graph
def printGraph(graph,n):
    for row in range(n):
        for column in range(n):
            print("{:.5f}".format(graph[row*n+column]), end = '\t')
        print()

# Eva requires special input, this function prepares the eva input
# Eva will then encrypt them
def prepareInput(n, vec_size):
    input = {}
    GG, nodes = generateGraph(n, 0.5)

```

```
    in_degree = [0] * n
    for (src, dest) in nodes:
        in_degree[dest] += 1

    for i in range(vec_size-n):
        in_degree.append(0)

    # graph is a list
    graph, graphdict = serializeGraphZeroOne(GG, vec_size)
    input['Graph'] = graph
    return input, in_degree

global_state = "start"
global_in_degree = []
global_graph_size = 0
global_vector_size = 0

def graphanalticprogram(graph):
    global global_graph_size
    global global_vector_size
    global global_in_degree
    global global_state

    # Array for multiplication
    first_one = []
    for i in range(global_vector_size):
        first_one.append(0.0)
    first_one[0] = 1

    node_queue = []
    for i in range(global_graph_size):
        if global_in_degree[i] == 0:
            node_queue.append(i)

    top_order = []

    while node_queue:
        # Extract front of queue (or perform dequeue) and add it to topological order
        node_index = node_queue.pop(0)
        top_order.append(node_index)

        node_dest_block = graph << (node_index*global_graph_size) # Shifting
        for k in range(global_graph_size):
            current_dest = node_dest_block << k
            dest_exists_checker = current_dest * first_one # Multiplication
            global_state = "check_if_all_zero"

            yield global_state, dest_exists_checker # yielding!!!

            if global_state == "not_all_zero": # There is an edge from u to k
                global_in_degree[k] = global_in_degree[k] - 1
                if global_in_degree[k] == 0:
                    node_queue.append(k)

    yield "finished", top_order

# Do not change this
# the  parameter n can be passed in the call from simulate function
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)
```

```
# Repeat the experiments and show averages with confidence intervals
# You can modify the input parameters
# n is the number of nodes in your graph
# If you require additional parameters, add them
def simulate(n):
    global global_state
    global global_in_degree
    global global_graph_size
    global global_vector_size

    results = {
        "Iterations": [],
    }

    m = 256

    print("Will start simulation for ", n)
    config = {}
    config['warn_vec_size'] = 'false'
    config['lazy_relinearize'] = 'true'
    config['rescaler'] = 'always'
    config['balance_reductions'] = 'true'
    inputs, global_in_degree = prepareInput(n, m)
    global_graph_size = n
    global_vector_size = m

    graphanaltic = EvaProgramDriver("graphanaltic", vec_size=m,n=n)
    with graphanaltic:

        total_start = timeit.default_timer()
        # Graph adjacency list is private information, therefore we will encrpyt it
        # In degree and discovered lists are computational helpers and they are not
    encrypted.
        graph = Input('Graph') # Encrypted input
        p = graphanalticprogram(graph)
        iter = 0
        prog = graphanaltic
        compiler = CKKSCompiler(config=config)
        while True:
            try:
                global_state, reval = next(p)
            except StopIteration as e:
                print("StopIteration exception occured. Breaking the loop...", e)
                break

            if global_state == "finished":
                print("Sorting is finished")
                print("Topological sort:", reval)
                break

            Output(f'ReturnedValue{iter}', reval)

            if global_state == "check_if_all_zero":
                result = {}
                prog.set_output_ranges(30)
                prog.set_input_scales(30)

                iter_start = timeit.default_timer()
                compiled_multfunc, params, signature = compiler.compile(prog)
                compiletime = (timeit.default_timer() - iter_start) * 1000.0 #ms
                result["CompileTime"] = compiletime

                iter_start = timeit.default_timer()
                public_ctx, secret_ctx = generate_keys(params)
                keygenerationtime = (timeit.default_timer() - iter_start) * 1000.0 #
    ms
                result["KeyGenerationTime"] = keygenerationtime
```

```
                iter_start = timeit.default_timer()
                encInputs = public_ctx.encrypt(inputs, signature)
                encryptiontime = (timeit.default_timer() - iter_start) * 1000.0 #ms
                result["EncryptionTime"] = encryptiontime

                iter_start = timeit.default_timer()
                encOutputs = public_ctx.execute(compiled_multfunc, encInputs)
                executiontime = (timeit.default_timer() - iter_start) * 1000.0 #ms
                result["ExecutionTime"] = executiontime

                iter_start = timeit.default_timer()
                outputs = secret_ctx.decrypt(encOutputs, signature)
                decryptiontime = (timeit.default_timer() - iter_start) * 1000.0 #ms
                result["DecryptionTime"] = decryptiontime

                iter_start = timeit.default_timer()
                reference = evaluate(compiled_multfunc, inputs)
                referenceexecutiontime = (timeit.default_timer() - iter_start) * 1000.0 #ms
                result["ReferenceExecutionTime"] = referenceexecutiontime

                if [k for k in reference[f'ReturnedValue{iter}'][:n*n] if k!=0]: # If not all zeros
                    global_state = "not_all_zero"
                else:
                    global_state = "all_zero"

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

                results["Iterations"].append(result)
            iter += 1

    results["TotalTime"] = (timeit.default_timer() - total_start) * 1000.0 #ms
    return results


if __name__ == "__main__":
    total_results = []
    bm_keys = ["Sim#", "NodeCount","TotalIteration","CompileTime","KeyGenerationTime","EncryptionTime","ExecutionTime","DecryptionTime","ReferenceExecutionTime"]

    simcnt = 6 #The number of simulation runs, set it to 3 during development otherwise you will wait for a long time

    try:
        mkdir("results")
    except:
        pass

    resultfile = open("results/results.csv", "w")  # Measurement results are collated in this file for you to plot later on
    resultfile.write(",".join(bm_keys) + "\n")
    resultfile.close()

    print("Simulation campaing started:")
    for nc in range(6, 18, 4): # Node counts for experimenting various graph sizes
        n = nc
        resultfile = open("results/results.csv", "a")

        # Dict to hold the data to plot
        total_res = {}
        total_res["NodeCount"] = n
        total_res["CompileTime"] = []
        total_res["KeyGenerationTime"] = []
        total_res["EncryptionTime"] = []
```

```
        total_res["ExecutionTime"] = []
        total_res["DecryptionTime"] = []
        total_res["ReferenceExecutionTime"] = []

        for i in range(simcnt):
            total_res["CompileTime"].append(0)
            total_res["KeyGenerationTime"].append(0)
            total_res["EncryptionTime"].append(0)
            total_res["ExecutionTime"].append(0)
            total_res["DecryptionTime"].append(0)
            total_res["ReferenceExecutionTime"].append(0)

            #Call the simulator
            results = simulate(n)
            total_res["TotalTime"] = results["TotalTime"]
            iteration = 0
            for result in results["Iterations"]:
                iteration += 1
                total_res["CompileTime"][i] += result["CompileTime"]
                total_res["KeyGenerationTime"][i] += result["KeyGenerationTime"]
                total_res["EncryptionTime"][i] += result["EncryptionTime"]
                total_res["ExecutionTime"][i] += result["ExecutionTime"]
                total_res["DecryptionTime"][i] += result["DecryptionTime"]
                total_res["ReferenceExecutionTime"][i] += result["ReferenceExecutionTime"]

            csv_res = str(i) + "," + str(n) + "," + str(len(results["Iterations"])) + "," + str(total_res["CompileTime"][i]) + "," + str(total_res["KeyGenerationTime"][i]) + "," +  str(total_res["EncryptionTime"][i]) + "," +  str(total_res["ExecutionTime"][i]) + "," +  str(total_res["DecryptionTime"][i]) + "," +  str(total_res["ReferenceExecutionTime"][i]) + "\n"
            resultfile.write(csv_res)
        total_results.append(copy.deepcopy(total_res))

        resultfile.close()

    for key in [k for k in bm_keys if k not in ["Sim#", "NodeCount","TotalIteration", "Mse"]]:
        plt.cla()
        for res in total_results:
            if key in res:
                plt.plot(list(range(simcnt)), res[key], label = f"Node count {res['NodeCount']}")
        # plt.xlabel("PathLength")
        plt.ylabel(key)
        plt.title(key)
        plt.grid()
        plt.legend()
        plt.autoscale(enable=True, axis="y", tight=None)
        plt.savefig(f"results/{key}.png")
        # plt.show(block=False)
```