Homomorphic Encryption: Conway's Game of Life meets the Simple Encrypted Arithmetic Library (SEAL)

Patrick Tu and Kathrin Witzlsperger

Project Documentation
Data Science and Ethics
Ludwig-Maximilians-Universität München
by
Patrick Tu and Kathrin Witzlsperger
Munich, 15th July 2018

Contents

Chapter 1

Introduction

At least since the introduction of the *General Data Protection Regulation (GDPR)* in the European Union on May 25th 2018, data protection and data privacy are ubiquitous and frequently discussed topics. This shows how important the security of private or confidential data is to both individuals and corporates. Furthermore, globally reported data breaches such as the *Facebook - Cambridge Analytica data scandal* [1] created an additional consumer sensitivity about where their data is processed and how well it is protected. As today data and lots of computationally expensive operations are processed on remote servers on a daily basis, the need for secure and privacy-maintaining encryption schemes is getting more and more important in order to satisfy not only imposed laws such as the *GDPR* but also the individual's need for privacy and anonymity.
One approach to tackle this need is homomorphic encryption that entails the big advantage that computations can be directly performed on encrypted data [2]. This means that, while making use of the computational power of remote servers in the cloud, the data is still encrypted and protected. Hence, privacy and confidentiality can be ensured while interacting with possibly untrusted environments [3]. However, one major drawback is that running these encrypted operations is very time-consuming and memory-intense. That's why improving the performance of homomorphic encryption and developing new fully homomorphic encryption schemes, that allow an arbitrarily ordered and unlimited number of operations, is a very active research area with huge interest due to numerous potential real world applications [2].

The goal of this work is to implement Conway's Game of Life using homomorphic encryption for any operation using the *Simple Encrypted Arithmetic Library* (*SEAL*, see `http://sealcrypto.org/`) that is maintained by the Cryptography Research Group at Microsoft Research and implements a fully homomorphic encryption scheme in C++ [4]. To do so, chapter 2 first briefly reviews some theoretical background regarding the Game of Life and homomorphic encryption. Then, chapter 3 presents the concept as well as the implementation of this work. Last but not least chapter 4 will conclude with a discussion touching upon the results, challenges and conclusions.

Chapter 2

Theoretical Background

2.1 Conway's Game of Life

Conway's Game of Life is a cellular automaton developed by the British mathematician John Horton Conway in 1970. It is a zero-player game where based on an initial configuration of a rectangular two-dimensional grid, the board evolves based on predefined rules [5]. In this potentially infinite, orthogonal grid of squares each cell can have one of two possible states: *dead* or *alive*. Every cell has eight neighbor cells that are horizontally, vertically or diagonally adjacent to it. There are four rules that determine the state of a cell in the next step (the next generation) based on its own state and the states of its neighbors:

- **Underpopulation**
  Any live cell with fewer than two live neighbors dies.
- **Next Generation**
  Any live cell with two or three live neighbors lives on to the next generation.
- **Overpopulation**
  Any live cell with more than three live neighbors dies.
- **Reproduction**
  Any dead cell with exactly three live neighbors becomes a live cell.

The first generation is created by simultaneously applying all of these rules to each cell of the initial grid state, which means that deaths and births occur simultaneously in a discrete moment. To generate further generations, one applies the rules repeatedly to the latest grid states. Hence, in the life cycle of the Game of Life each generation depends on the previous one [6].

2.2 Homomorphic Encryption

Homomorphic encryption refers to a special type of encryption technique that allows specific computations to be performed on ciphertexts and generate an encrypted result, which, when decrypted, matches the result of those operations performed on the corresponding plaintexts [7].
It draws on the mathematical concept of group homomorphism, a structure preserving map between two algebraic groups. A group is a set, G, together with a certain operation $\circ$ (called the group law of G) that combines any two elements a and b to form another element, denoted by $a \circ b$. To qualify as a group, the set and operation, $(G; \circ)$, must satisfy four requirements known as the group axioms [8]:

- **Closure**
  $\forall a, b \in G : a \circ b \in G$
- **Associativity**
  $\forall a, b, c \in G$: $(a \circ b) \circ c = a \circ (b \circ c)$
- **Identity**
  $\exists e \in G : \forall a \in G : e \circ a = a = a \circ e$
- **Inverse**
  $\forall a \in G : \exists b \in G : a \circ b = e = b \circ a$

Given two groups, $(G, \diamond)$ and $(H, \circ)$, a group homomorphism from $(G; \diamond)$ to $(H; \circ)$ is a function $f : G \to H$ such that $\forall a, b \in G$ it holds that:

$$f(a \diamond b) = f(a) \circ f(b)$$

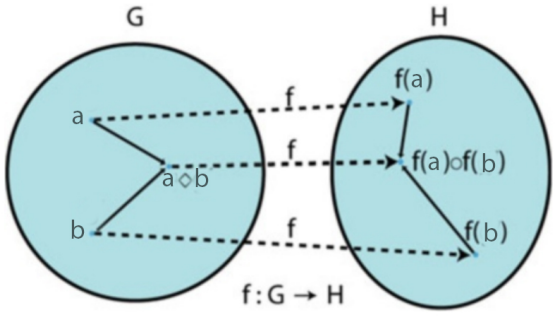

Figure 2.1: Group homomorphism [8]

This concept can be transferred to an encryption scheme (P, C, K, E, D). P denotes the plaintext, C the ciphertext, K the key space, E the encryption algorithm and D the decryption algorithm. By assuming that the plaintext forms a group $(P, \diamond)$ and the ciphertext a group $(C, \circ)$, the encryption algorithm can be seen as a map $E_k : P \to C$. The encryption scheme (P, C, K, E, D) is called **homomorphic** if $\forall a \in P$ and $\forall k \in K$ the following equality holds:

$$E_k(a) \circ E_k(b) = E_k(a \diamond b)$$

Informally, such a scheme allows Alice, given her plaintext P, to compute the encryption $E_k(P)$ based on a key k and send it to Bob. Bob can then perform operations on the encrypted message without ever getting the key k and hence without ever learning anything about the initial plaintext P or $E_k(P)$ [8].

Chapter 3

Approach

3.1 Idea

The goal of this work is to use homomorphic encryption on Conway's Game of Life. Following the idea and big advantage of homomorphic encryption that operations can be performed on a remote location without the need of decrypting the data first, it seems obvious to choose a client-server architecture. While the client is responsible for the encryption and decryption of relevant data that is exchanged with the server as well as the display of the current game state in a graphical user interface (GUI), the server performs the operation of determining the next game state that corresponds to the Game of Life's main logic that was described in section 2.1.
Figure 3.1 visualizes the basic concept of the client-server model. To ensure confidentiality of the data, the idea is to encrypt every generation t homomorphically and hand it over to the server afterwards. The server then performs operations on the encrypted data to determine the following grid state of generation $t+1$ and hands it back to the client. After receiving the encrypted generation $t+1$, the client decrypts and displays the new grid state in the GUI.

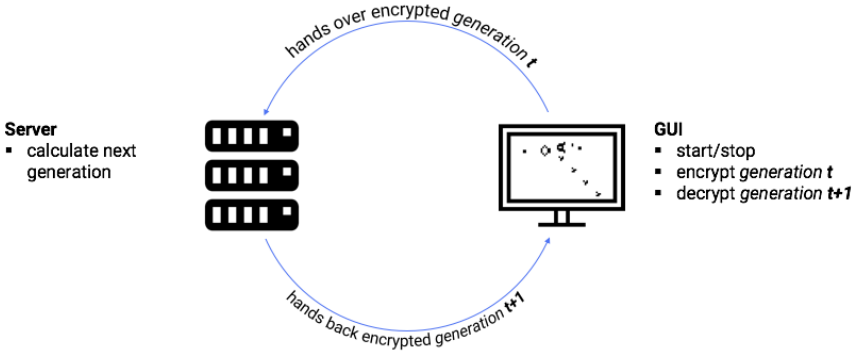

Figure 3.1: Client-server concept for Conway's Game of Life using homomorphic encryption

3.2 Concept

As the grid state of generation $t+1$ is dependent on the update rules introduced in section 2.1 that are usually implemented using simple if-else statements, it was necessary to come up with an intermediate step such that the standard operations (multiplication, addition) implemented in the *SEAL* [4] could be used to determine the next generation by only working on the encrypted data. To do so, an additional encoding scheme on top of the homomorphic encryption was developed in order to be able to perform the computation of the next grid state by just using the add-operation provided by the *SEAL*. As the state $t+1$ of a cell is dependent on both its own state and the number of neighbors, both information need to be encoded and considered in the process. The different steps of the concept that set the base for the implementation, presented in section 3.3, are illustrated in the following.

3.2.1 Encryption of current grid generation

The state of the Game of Life grid at generation t can be denoted by the matrix $A^{(t)}$, where the state of cell $a_{i,j} \in \{0,1\}$ encodes the cell being *dead* or *alive*. This is the first component required for the computation outsourced to the server. The matrix $A^{(t)}$ is encrypted by applying the built-in homomorphic encryption function of the *SEAL* to every element $a_{i,j}$ of generation t in order to get the encrypted grid matrix $A'^{(t)}$ that contains the ciphertext objects.

$$A^{(t)} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \xrightarrow{\text{elementwise encryption}} A'^{(t)} = \begin{pmatrix} a'_{1,1} & \cdots & a'_{1,n} \\ \vdots & \ddots & \vdots \\ a'_{n,1} & \cdots & a'_{n,n} \end{pmatrix}$$

3.2.2 Computation of neighbor matrix

As the second component to obtain the grid state $t+1$ is the number of live neighbors for each cell, these information need to be computed. For each cell $a_{i,j}$, the number of adjacent live cells are counted, captured and stored in the neighbor matrix $N^{(t)}$. Within the neighbor matrix, $n_{i,j}$ denotes the number of live neighbors of cell $a_{i,j}$. To consider neighboring cells that would be outside of the grid's range, it is assumed, that the left and right as well as the top and bottom edges of the grid are stitched together, yielding a toroidal array. In this way all cells obtain eight neighbors.

$$A^{(t)} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \xrightarrow{\text{compute neighbors}} N^{(t)} = \begin{pmatrix} n_{1,1} & \cdots & n_{1,n} \\ \vdots & \ddots & \vdots \\ n_{n,1} & \cdots & n_{n,n} \end{pmatrix}$$

3.2.3 Encoding and encryption of neighbor matrix

In this step, the neighbor matrix $N^{(t)}$ is encoded in a way such that by pairwise addition with the initial board state $A^{(t)}$ it can be decoded to the game state $A^{(t+1)}$ using some predefined rules presented in subsection 3.2.5. In an encryption context these encoding and decoding rules could be seen as the keys that just the client possesses.
The neighbor matrix $N^{(t)}$ is encoded by applying the following encoding function f to each element $n_{i,j}$:

$$f(n_{i,j}) = \begin{cases} 0 & \text{for } n_{i,j} = 2 \\ 2 & \text{for } n_{i,j} = 3 \\ -2 & \text{otherwise} \end{cases}$$

This results in the encoded neighbor matrix $B^{(t)}$, where each element $b_{i,j}$ is computed by $f(n_{i,j})$. $B^{(t)}$ is afterwards encrypted homomorphically by applying the element-wise encryption function of the *SEAL* in the same way as in the first step presented in subsection 3.2.1, which results in the encrypted and encoded neighbor matrix $B'^{(t)}$.

$$N^{(t)} = \begin{pmatrix} n_{1,1} & \cdots & n_{1,n} \\ \vdots & \ddots & \vdots \\ n_{n,1} & \cdots & n_{n,n} \end{pmatrix} \xrightarrow{\substack{\text{encode} \\ \text{neighbors} \\ f(n_{i,j})}} B^{(t)} = \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,n} \end{pmatrix} \xrightarrow{\substack{\text{elementwise} \\ \text{encryption}}} B'^{(t)} = \begin{pmatrix} b'_{1,1} & \cdots & b'_{1,n} \\ \vdots & \ddots & \vdots \\ b'_{n,1} & \cdots & b'_{n,n} \end{pmatrix}$$

3.2.4 Server-side homomorphic add-operation

After the encryption process, the matrices $A'^{(t)}$ and $B'^{(t)}$ are transmitted to the server within the message $m = \{A'^{(t)}, B'^{(t)}\}$. The server then simply performs a matrix addition of the two transmitted matrices using pairwise addition with the built-in function of the *SEAL*.
$C'^{(t)} = A'^{(t)} + B'^{(t)}$ denotes the result of the matrix addition, where $c'_{i,j} = a'_{i,j} + b'_{i,j}$ is still in an homomorphically encrypted state. After this operation the result matrix $C'^{(t)}$ will be transmitted back to the client.

$$A'^{(t)} = \begin{pmatrix} a'_{1,1} & \cdots & a'_{1,n} \\ \vdots & \ddots & \vdots \\ a'_{n,1} & \cdots & a'_{n,n} \end{pmatrix} + B'^{(t)} = \begin{pmatrix} b'_{1,1} & \cdots & b'_{1,n} \\ \vdots & \ddots & \vdots \\ b'_{n,1} & \cdots & b'_{n,n} \end{pmatrix} \xrightarrow{\text{matrix addition}} C'^{(t)} = \begin{pmatrix} c_{1,1} & \cdots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{n,1} & \cdots & c_{n,n} \end{pmatrix}$$

3.2.5 Decryption and decoding of server response

Once the client receives the server response $C'^{(t)}$, it uses the *SEAL* to decrypt the result of the matrix addition. In this way the ciphertexts are converted to numerical values that are stored in the matrix $C^{(t)}$.
To finally get the state of the grid of generation $t+1$, the following decoding function g needs to be applied to each element of $C^{(t)}$:

$$g(c_{i,j}) = \begin{cases} 1 & \text{for } c_{i,j} > 0 \\ 0 & \text{for } c_{i,j} \leq 0 \end{cases}$$

This results in a matrix $A^{(t+1)}$, where each element is either 0 or 1 representing the state of a cell in generation $t+1$ in line with the rules of Conway's Game of Life that were introduced in section 2.1.

$$C'^{(t)} = \begin{pmatrix} c'_{1,1} & \cdots & c'_{1,n} \\ \vdots & \ddots & \vdots \\ c'_{n,1} & \cdots & c'_{n,n} \end{pmatrix} \xrightarrow{\text{elementwise decryption}} C^{(t)} = \begin{pmatrix} c_{1,1} & \cdots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{n,1} & \cdots & c_{n,n} \end{pmatrix} \xrightarrow{\text{elementwise decoding } g(c_{i,j})} A^{(t+1)} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}$$

3.3 Implementation

Due to our knowledge and expertise limitation of C++, we decided to implement the concept explained in section 3.2 using *PySEAL* (see `https://github.com/Lab41/PySEAL`) instead of the *SEAL*. *PySEAL* provides a dockerized Python wrapper implementation of the *SEALv2.3* using *pybind11* [9]. By building an image from the Dockerfile provided within *PySEAL* the Python package *seal* is created. This can then simply be imported and used within the also dockerized client-server model implemented in Python.

3.3.1 Client

The Python client is implemented in the following files:

- `client.py` realizes the GUI shown in figure 3.2 that has been implemented using *tkinter* and *threading* to allow to start and stop the animation of Conway's Game of Life by clicking on the corresponding buttons. A canvas widget is used to display the grid's states. Furthermore, a checkbox enables to run the application either with or without homomorphic encryption.
- `gameOfLife.py` models the Game of Life board as a *numpy* array of size 15×15 and randomly initializes cells *alive* and *dead*. If homomorphic encryption is enabled, it coordinates the update step of the current grid described in section 3.2, so that the server only needs to take care of one matrix addition. If homomorphic encryption

is not enabled, it sends the serialized current grid state to the server and finally deserializes the next grid state computed on the server.

- `encryption.py` summarizes all encryption, decryption, encoding and decoding methods needed within the implementation. Apart from the element-wise homomorphic encryption using the *seal* package, it further contains the encoding and decoding rules discussed in subsection 3.2.

- `helper.py` contains helper methods for the communication between client and server, which can be used by both the server and the client.

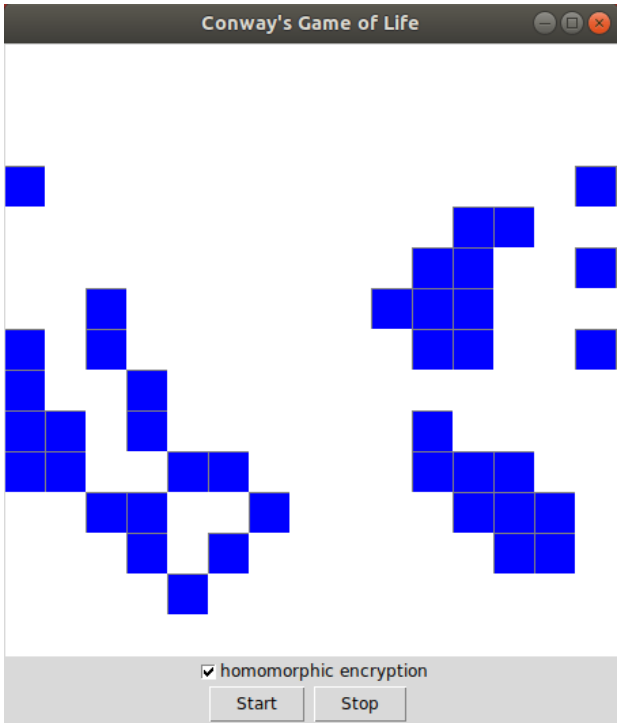

Figure 3.2: Graphical User Interface

3.3.2 Server

The Python server is realized in the following files:

- `server.py` contains the server implementation using TCP/IP sockets. It computes the next state of the grid in the **unencrypted case**. The server listens to a socket and starts a thread once a connection is accepted. Then, the server receives a grid configuration from the client as a serialized array, deserializes it and further computes the next generation $t+1$ based on the basic rules described in section 2.1. Afterwards, the new grid generation is transmitted back to the client as a serialized byte stream.

- `server_simulation.py` simulates a server using a thread and a message queue. This is used to compute the next grid generation in the **encrypted case**. The server thread waits for incoming messages using a message queue, fetches the encrypted message and then runs the operations described in subsection 3.2.4 using the *seal* package to compute the next grid state. This workaround using threads was chosen due to serialization issues with *PySEAL* objects, that will be further described in section 4.2.2.

- `helper.py` contains helper methods for the communication between client and server, which can be used by both the server and the client.

3.3.3 How to execute

To make this project (see `https://github.com/patricktu2/GameOfLifeMeetsPySEAL`), run it is necessary to have *Docker* (see `https://www.docker.com/`) as well as XQuartz (only needed for macOS, see `https://www.xquartz.org`) installed.
Afterwards you should proceed as follows:

1. Execute the `build.sh` script to build the image from the Dockerfile. This also builds the Python wrapper for the *SEAL*.

2. Execute the `run.sh` script to run a container derived from the previously built image and allow the GUI to connect to the host's display. This script consists of two parts as different solutions for *macOS* and *Ubuntu 18.04* were needed.

3. (optionally) Execute the `debug.sh` script to copy the log files of the server and the client from the container to the host. This can be used for debugging.

Chapter 4

Discussion

4.1 Results

Our main result is an implementation of Conway's Game of Life with a simple GUI, where the calculation of the next grid state can be homomorphically encrypted using *PySEAL*. As the efficiency of homomorphic encryption is often discussed in literature (see e.g. [2]), we furthermore decided to have a closer look at this issue within our project by comparing the runtime development of the Game of Life's update operation with and without homomorphic encryption. Our results are summarized in figure 4.1. Each point represents the median time measured for one update step. The median was selected by looking at ten measurements for every grid size that was considered. The time measurements were performed using the *timeit* package on a `Dell Inspiron 14-7437` with an `Intel Core i7-4510U` and `8GB RAM`.

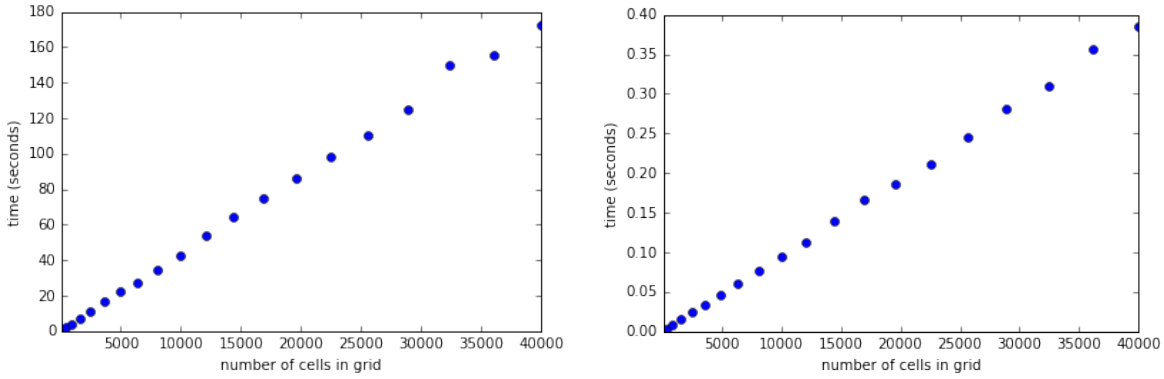

Figure 4.1: Comparison of runtime development of a single update operation for different grid sizes with (left) and without (right) homomorphic encryption

As expected, in both cases with rising grid size the time to compute one single update step increased. Furthermore, it can be seen that there exists a linear relationship between runtime and number of cells in the grid. In addition to that, it became obvious, that

independent of the grid size it always took significantly longer to run the game with homomorphic encryption than without. One step in a small encrypted 10×10 grid took even longer than updating a 200×200 grid without encryption. This result clearly shows that one of the major drawbacks of homomorphic encryption even exists for simple problems such as Conway's Game of Life. To make it usable for more practical and complex applications, it will be important to resolve this issue in the future.

4.2 Challenges

There were multiple challenging aspects involved in the implementation of this project that are presented in the following subsections.

4.2.1 Docker

As we decided to use *PySEAL* instead of the *SEAL*, it was necessary to work with *Docker*. *Windows 10 Home* does not support Hyper-V, therefore a separate *Linux* distribution, namely *Ubuntu 18.04*, was set up to be able to install and make *Docker* run on one of our machines. The other one based on *macOS* didn't face this issue. A further challenge was to jointly and collaboratively develop an application using *Docker* on two different operating systems. As already mentioned in section 3.3.1 the display of the GUI from within a *Docker* container needed different solutions such as installing third-party software on *macOS*. Furthermore, developing with *Docker* appeared to us more time consuming than expected. Due to our non-existent previous experience with *Docker* it was a special challenge to make the client and server run in parallel within our *Docker* container. To solve this issue we used *tmux*, a terminal multiplexer (see `https://github.com/tmux/tmux`). This led to another problem regarding debugging, as in this way the console outputs of the server and client were no longer printed to the terminal. We circumvented this problem by redirecting the console outputs to two log files that can be inspected after copying them from the container to the host using the `debug.sh` script.

4.2.2 Serialization

The serialization of encrypted ciphertexts was another challenge we faced. After building the client-server concept for our base case, which performed the update computation of the grid without homomorphic encryption, we realized that the necessary *PySEAL* instances (in particular `Ciphertext` and `Context`) were not serializable. We tested different packages such as *pickle*, *byteIO* and *dill*, but unfortunately the serialization didn't work out. In a GitHub issue that was raised concerning this problem (see `https://github.com/Lab41/PySEAL/issues/16`), a developer of the project commented that "this may be impossible due to native dependencies". Moreover, we tried to expose the built-in `save(std::ostream stream)` and `load(std::istream stream)` function of the `Ciphertext` class of the initial C++ *SEAL* using *pybind11*, as they seemed to save/load a ciphertext to/from a binary

output stream/input stream. But this led to the issue that there is no implicit type conversion from the Python `io` to the C++ `iostream` class possible. Due to these problems and the limited time window, we finally decided to simulate the server, which runs the encrypted operations using *PySEAL*, by means of threads and message queues.

4.3 Conclusion

This work demonstrates homomorphic encryption using the simple example of Conway's Game of Life. While in this context the server was simulated locally on the same machine, the practical use cases and implications still became clear.

In times where data is an important asset that can contain sensitive and confidential information, companies and individuals highly mind where their data is processed. Homomorphic encryption could be a way to make use of remote computing power, while still ensuring the privacy of the data. This could have industry-wide applications and implications, especially for economies or societies that are very sensitive towards data privacy and protection such as Germany. However, the main drawbacks of homomorphic encryption are performance-wise. As it was shown in section 4.1 the computing time for encrypted operations increases significantly with the size of the data. IBM faced the same issue with their first attempts at homomorphic encryption, which ran 100 trillion times slower than the corresponding computations on plaintexts [10]. Hence, outsourcing the computation using homomorphic encryption would just make sense when encrypting inputs and decrypting outputs is faster than performing the computation itself. In a recent release of IBM's library *HElib* (see `https://github.com/shaih/HElib`) a significant improvement of performance was reached [11], but for an enterprise- and industry-wide adoption of this technology further developments regarding efficiency still need to be taken. Besides that, most implementations are just partially homomorphic cryptosystems meaning that not all, arbitrary computations on ciphertexts are supported [5]. Nevertheless, homomorphic encryption is a very promising field. Further developments in fully homomorphic encryption, which is still in research phase, could overcome security concerns of cloud computing and enable secure applications, storage and services to be offered regardless of where the servers reside [12].

Appendix A

Encoding and Decoding Example

This small example based on a 6×6 grid shows the encoding and decoding logic derived within this project that was presented in section 3.2.3 and 3.2.5.

Matrix $A^{(t)}$ denotes the state of the Game of Life grid at generation t, where the state of cell $a_{i,j}$ encodes the cell being *dead* $(= 0)$ or *alive* $(= 1)$.
To get the board state $t + 1$ for each cell $a_{i,j}$ the number of live neighbors needs to be computed. This information is stored in the neighbor matrix $N^{(t)}$, where $n_{i,j}$ denotes the number of live neighbors of cell $a_{i,j}$.

$$A^{(t)} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \xrightarrow{\text{compute neighbors}} N^{(t)} = \begin{pmatrix} 3 & 3 & 2 & 2 & 1 & 0 \\ 3 & 4 & 5 & 2 & 1 & 1 \\ 4 & 3 & 5 & 5 & 3 & 2 \\ 2 & 4 & 4 & 4 & 3 & 2 \\ 2 & 2 & 4 & 3 & 3 & 1 \\ 1 & 1 & 2 & 1 & 1 & 0 \end{pmatrix}$$

As mentioned in section 3.2.3 the neighbor matrix $N^{(t)}$ needs to be encoded by applying the following encoding function f to each element $n_{i,j}$:

$$f(n_{i,j}) = \begin{cases} 0 & \text{for } n_{i,j} = 2 \\ 2 & \text{for } n_{i,j} = 3 \\ -2 & \text{otherwise} \end{cases}$$

This results in the encoded neighbor matrix $B^{(t)}$, where each element $b_{i,j}$ is computed by $f(n_{i,j})$.

$$N^{(t)} = \begin{pmatrix} 3 & 3 & 2 & 2 & 1 & 0 \\ 3 & 4 & 5 & 2 & 1 & 1 \\ 4 & 3 & 5 & 5 & 3 & 2 \\ 2 & 4 & 4 & 4 & 3 & 2 \\ 2 & 2 & 4 & 3 & 3 & 1 \\ 1 & 1 & 2 & 1 & 1 & 0 \end{pmatrix} \xrightarrow{\text{encode neighbors } f(n_{i,j})} B^{(t)} = \begin{pmatrix} 2 & 2 & 0 & 0 & -2 & -2 \\ 2 & -2 & -2 & 0 & -2 & -2 \\ -2 & 2 & -2 & -2 & 2 & 0 \\ 0 & -2 & -2 & -2 & 2 & 0 \\ 0 & 0 & -2 & 2 & 2 & -2 \\ -2 & -2 & 0 & -2 & -2 & -2 \end{pmatrix}$$

$C^{(t)} = A^{(t)} + B^{(t)}$ denotes the result of the matrix addition that is actually performed using homomorphic encryption.

$$A^{(t)} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} + B^{(t)} = \begin{pmatrix} 2 & 2 & 0 & 0 & -2 & -2 \\ 2 & -2 & -2 & 0 & -2 & -2 \\ -2 & 2 & -2 & -2 & 2 & 0 \\ 0 & -2 & -2 & -2 & 2 & 0 \\ 0 & 0 & -2 & 2 & 2 & -2 \\ -2 & -2 & 0 & -2 & -2 & -2 \end{pmatrix} \xrightarrow{\text{matrix addition}} C^{(t)} = \begin{pmatrix} 2 & 3 & 1 & 0 & -2 & -2 \\ 3 & -1 & -2 & 1 & -2 & -2 \\ -2 & 3 & -2 & -2 & 3 & 0 \\ 1 & -2 & -1 & -1 & 3 & 0 \\ 0 & 1 & -2 & 3 & 2 & -2 \\ -2 & -2 & 0 & -2 & -2 & -2 \end{pmatrix}$$

To finally get the state of the grid of generation $t+1$, the following decoding function g is applied to each element of $C^{(t)}$:

$$g(c_{i,j}) = \begin{cases} 1 & \text{for } c_{i,j} > 0 \\ 0 & \text{for } c_{i,j} \leq 0 \end{cases}$$

This results in a matrix $A^{(t+1)}$, where each element is either 0 or 1, representing the state of the next generation $t+1$ in line with the rules of Conway's Game of Life that were presented in section 2.1.

$$C^{(t)} = \begin{pmatrix} 2 & 3 & 1 & 0 & -2 & -2 \\ 3 & -1 & -2 & 1 & -2 & -2 \\ -2 & 3 & -2 & -2 & 3 & 0 \\ 1 & -2 & -1 & -1 & 3 & 0 \\ 0 & 1 & -2 & 3 & 2 & -2 \\ -2 & -2 & 0 & -2 & -2 & -2 \end{pmatrix} \xrightarrow{\text{elementwise decoding } g(c_{i,j})} A^{(t+1)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Bibliography

[1] Wikipedia, "Facebook-Cambridge Analytica data scandal — Wikipedia, The Free Encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal

[2] Y. Hu, "Improving the efficiency of homomorphic encryption schemes," Ph.D. dissertation, Worcester Polytechnic Institute, 2013.

[3] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A Guide to Fully Homomorphic Encryption," *IACR Cryptology ePrint Archive*, p. 1192, 2015.

[4] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library SEAL v2.2," Tech. Rep. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/sealmanual_v2.2.pdf

[5] Wikipedia, "Conway's Game of Life — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Conway27s_Game_of_Life

[6] M. Gardner, "MATHEMATICAL GAMES," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970. [Online]. Available: http://www.jstor.org/stable/24927642

[7] M. Ogburn, C. Turner, and P. Dahal, "Homomorphic Encryption," pp. 502–509, 2013.

[8] X. Yi, R. Paulet, and E. Bertino, *Homomorphic encryption and applications*. Springer, 2014.

[9] A. J. Titus, S. Kishore, T. Stavish, S. M. Rogers, and K. Ni, "PySEAL: A Python wrapper implementation of the SEAL homomorphic encryption library," *arXiv preprint arXiv:1803.01891*, 2018. [Online]. Available: https://arxiv.org/pdf/1803.01891.pdf

[10] R. Chirgwin, "IBM's homomorphic encryption accelerated to run 75 times faster," 2018. [Online]. Available: https://www.theregister.co.uk/2018/03/08/ibm_faster_homomorphic_encryption/

[11] S. Halevi and V. Shoup, "Faster Homomorphic Linear Transformations in HElib," Cryptology ePrint Archive, Report 2018/244, 2018. [Online]. Available: https://eprint.iacr.org/2018/244

[12] A. Waller, "Homomorphic encryption: A guide to advances in the processing of encrypted data," 2016. [Online]. Available: https://www.thalesgroup.com/en/critical-information-systems-and-cybersecurity/news/homomorphic-encryption-guide-advances-processing