Grant Agreement Number: 825225

Safe-DEED

www.safe-deed.eu

Implementation of cryptographic building blocks and specialized protocols v2/3

Deliverable number	*D5.9*
Dissemination level	*Public*
Delivery data	*due 30.11.2020*
Status	*Final*
Authors	*Alexander Grass, Lukas Helminger, Fabian Schmid*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825225.

Changes Summary

Date	Author	Summary	Version
23.10.2020	Alexander Grass, Fabian Schmid	First Draft	0.1
30.10.2020	Alexander Grass, Lukas Helminger, Fabian Schmid	Second Draft (for internal review)	0.2
17.11.2020	Alexandros Bampoulidis, Evangelos Kotsifakos	Internal Review	0.3
20.11.2020	Final Version	Alexander Grass, Lukas Helminger, Fabian Schmid	1.0

Executive Summary

This deliverable - D5.9 Implementation of cryptographic building blocks and specialized protocols v2 - is the third software result of Safe-DEED's WP5 in the form of a demonstrator. It is an outcome of task T5.3 - Implementation of protocols, which were developed in task T5.1 - Private set intersection, secure multiparty computation, and privacy-preserving computation on data owned by multiple parties and task T5.2 - Specialized protocols.

This demonstrator consists of several implementations (C++, Python GUI, JavaScript) of an extremely versatile privacy-enhancing protocol called Private Selective Aggregation (PSA), see D5.8. This protocol's applications range from making the WP4's Data Valuation Component Questionnaire privacy-preserving to a COVID-19 heatmap.

In addition, this deliverable comprises a document that describes how to use, apply, and modify these software components. The provided demonstrator is a proof of concept. The plan is to integrate some of the functionality to the WP4's Data Valuation Component.

Table of Contents

List of Figures

Abbreviations

CDR	Call Detail Records
GUI	Graphical User Interface
PSA	Private Selective Aggregation

1 Introduction

This demonstrator aims to implement the privacy-enhancing protocol Private Selective Aggregation (PSA). PSA was co-developed within Safe-DEED. The specification, purpose, security, privacy as well as use-cases are extensively covered in Safe-DEED's deliverable D5.8. This document is only concerned with PSA's implementations. Most of the time, we will explain the implementations based on a life science use-case. Note that there are different use-cases. For example, one is concerned with privacy-preserving data analytics (WP4).

1.1 Synopsis of the Protocol

In the context of privacy-enhancing technologies, we sought to create reliable and secure corona heatmaps. Our goal was to compute and visualize the distribution of COVID-19 infected persons. We aimed to achieve it by the combination of data from health authorities and mobile network providers. However, the real challenge was to assert strong security guarantees both for the authorities and the operators. During development, we worked with public location data centered around Vienna.

It is a two-party protocol in the classical client-server setting. The Client (health authority) has the identity of patients. The Server (mobile network operator) has Call Detail Records (CDRs). We strive to output the aggregated location data from those CDR, which match the patient's identifiers. Naturally, we want to protect the identity of the ill and location data of individuals.

To achieve the privacy goals outlined above, we use homomorphic encryption, zero-knowledge proof techniques, and differential privacy.

In particular, the patients' identifiers get homomorphically encrypted before sending them to the mobile operator. Due to the nature of homomorphic encryption, the mobile operator can perform the data aggregation without decrypting the identifiers. To prevent the researchers from learning individual CDR, we ensure that the identifiers' set has a minimum cardinality by applying zero-knowledge proof techniques. The mobile operator can also add noise - in the sense of differential privacy - to the aggregated CDR before sending them to the researchers. This addition becomes necessary if the aggregated CDR still leak information that could lead to patients' re-identification.

1.2 Implementation's Structure

We divided the implementation of the protocol into multiple parts. On the one hand, we have a c++ implementation, representing our protocol's cryptographic backbone. On the other hand, we have a python implementation, which covers the graphical user interface. Moreover, we provide a JavaScript implementation to show that the protocol can be run in a browser.

All of these implementations have a server and a client application. The Client provides functionality to create a request and process a response from the server. The processed response is a heatmap of aggregated data. On the other hand, the server receives requests, performs the necessary operations, and responds with the encrypted aggregated information.

1.3 Road-map

In Section 2, we describe how to use this demonstrator. First, we explain how the Python GUI works (Section 2.1) and also how to run it (Section 2.1.3). We continue in Section 2.2 to describe the underlying C++ implementation. Lastly, we show how the corresponding JavaScript library can be operated (Section 2.3). For further details of these implementation we defer to Section 3.

2 Running the Protocols

This section aims to provide a general view of the application. We will discuss every step necessary to run and test our applications. This includes the build process for the c++ implementation, which is necessary to analyze differently structured datasets and the setup of NodeJS to simulate a web server and run the protocol in a browser.

As we have separated our program into two parts, we will also discuss them separately. The first part aims to convey our ideas to the general public. The second provides full functionality for technically experienced users.

Remark: The following instructions were written for Linux. More concretely, we tested the applications using Ubuntu 20.04. We tried to simplify the steps as far as possible, but one probably still needs some familiarity with software development.
Important data import: As we exceeded the file size limits, we outsourced some of the essential parts [1]. On Zenodo, you should find three important files. The client, the server and the at.sqlite file. From the root of our attachment, one needs to put the three files into these exact positions.

- `project_root/GUI_Executabe/client` should be the path for client
- `project_root/GUI_Executabe/server` should be the path for server
- `project_root/GUI_Executabe/data/at.sqlite` should be the first path of at.sqlite. Put another copy to the following location
- `project_root/pythonDemo/data/at.sqlite` This dataset contains cell tower information.

2.1 Python GUI

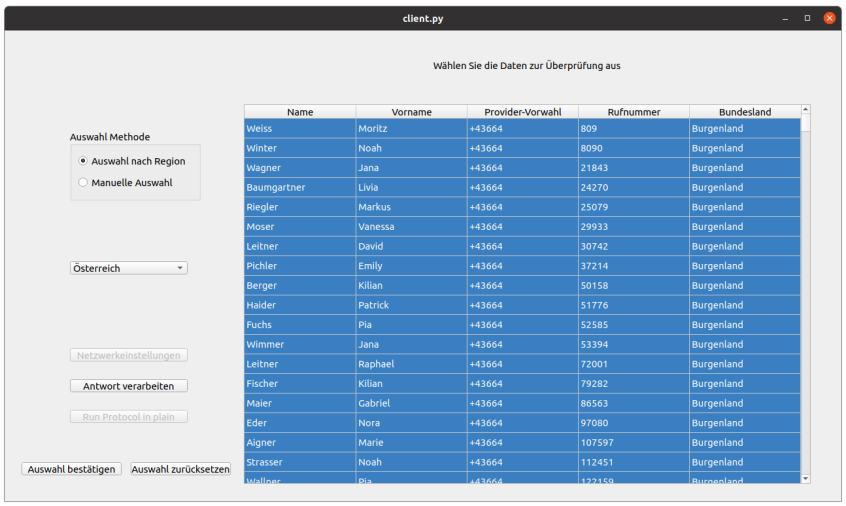
In addition to this deliverable, we provide the executables for both the server and the client. It is important not to change the directory structure. There should be a backend, a data, and a request folder in the executables' directory.

2.1.1 Step-by-Step Demonstration

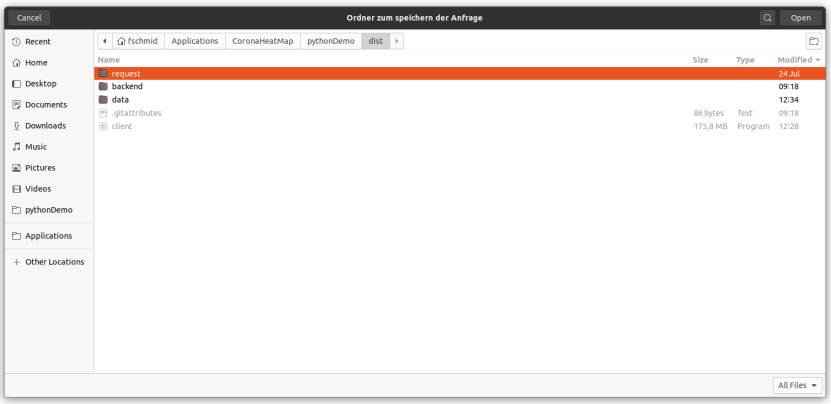
The protocol always starts with a request from the client. Therefore we start by presenting the client application.

[1] `https://zenodo.org/record/4293592#.X8D5RsIo_JU`

1. We start the application at the data selection window. In the precompiled user interface, we can only select patients from a predefined data set. The names and numbers do not represent actual people. However, each entry represents some anonymous individual from the Gowalla data set[2], centered around Vienna. We can select patients manually or select entire regions. It is worth mentioning that the number of selected people has a lower bound set in the c++ implementation. We will proceed by creating a request for all Austrian patients. Therefore we need to select the radio button ”Auswahl nach Region.”(select by region) Then, we select the ”Österreich”(Austria) option in the dropdown menu. To generate a request with the selected identifiers, we need to click the ”Auswahl bestätigen”(accept selection) button.

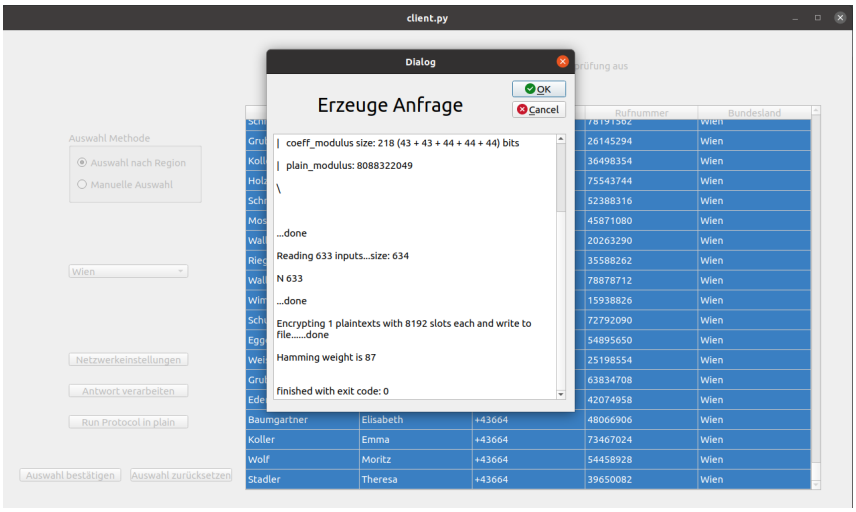


2. Before we can create a request, we need to select a location to store it. The request procedure will create five different files. In a real setting, the server should never access the files named ”rk” and ”gk.” For testing such a setting, we recommend copying the folder and deleting those files. This measure leads to a public and a private folder. We will discuss accessing the public folder when we get to the server part. For now let us just select the ”backend” folder.



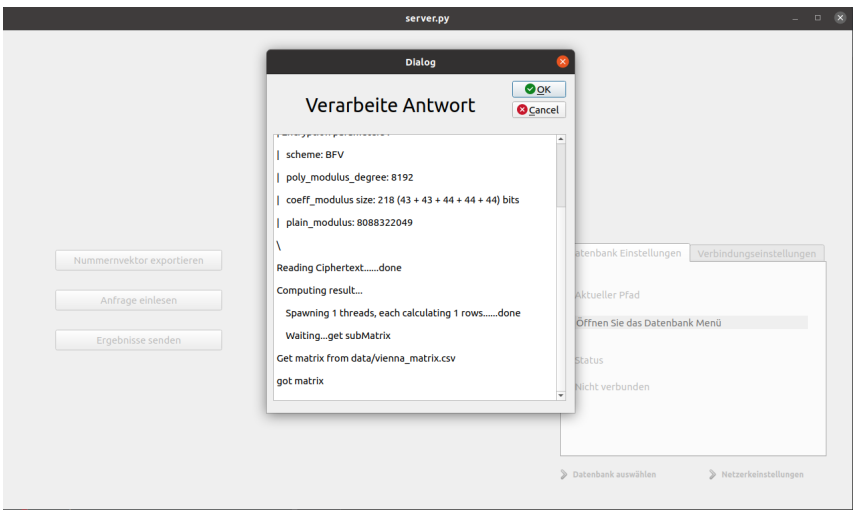
[2] https://snap.stanford.edu/data/loc-gowalla.html

3. After selecting a directory, the processing dialog opens. We can now read the output of the underlying cryptographic implementation. At this point, the application generates a vector with as many entries as there are patients. For each selected patient, the entry is one and zero for each non-selected patient. This vector is encrypted using homomorphic encryption, creating a ciphertext. Using homomorphic encryption allows the server to compute with the data while not learning anything about it.



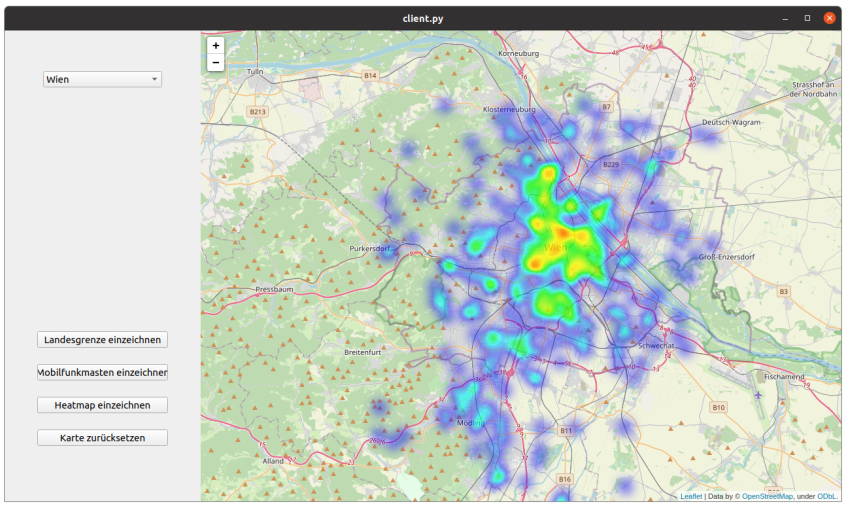
4. We have now successfully created the request. For the next step, we need to start the server application. If we want to start the server application on a different device, we need to ensure that we only share our request's public folder, as described above.

   In any case, to accept the request, we need to click the "Anfrage einlesen" button. Then a file browser will appear. There we select the folder (`backend`) of our request in order to create a response. Again, the output of the cryptographic implementation will show. We need to wait for the program to finish the computations. Depending on the computer, this might take several minutes. The server application stores the result in the "cipher_out" file in the same directory. If we run the server on a different device, we need to transfer the "cipher_out" file back to the client's private "request" directory.



5. We are now at a point where we can look at the result of our request. Therefore, we restart the client. In the client application, we click the ”Anfrage verarbeiten”(Process response) button and select the (`backend`) folder in the file browser. Again the processing dialog will open. In the background, the application is decrypting the response from the server.

   When the program in the background finishes, we can close the dialog (The application displays a return code in the dialog. Should be zero!) The next window displays a map centered in Vienna. By clicking the ”Heatmap einzeichen”(draw heatmap), we can display our selected data as a heatmap.



2.1.2 Additional Functionality

This section will briefly discuss the functionalities that we did not cover in the step-by-step guide. Let us take a look at the client first.

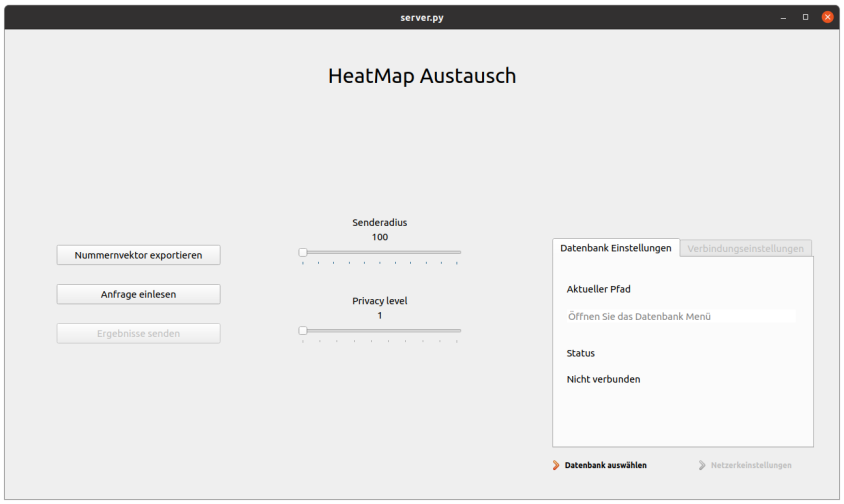
- Next to the selection by region, there is also a manual selection option. This feature is useful when dealing with small data sets. Since small datasets are not possible in a semi-trusted environment, we implemented it to test our resulting heatmap. By selecting a known pattern or single individuals, we could see direct results on the map.

- The two buttons, ”Netzwerkeinstellungen”(Network settings) and ”Run Protocol in Plain,” are currently disabled. Initially, we planned the transmission of data over the network. As we did not want to ramp up the trust assumptions, we decided to go for data transmission via USB-stick. Using this type of transmission, we ensure that both parties are always fully aware of all sent requests. This awareness further allows controlling the data flow.

  Running the protocol in plain allowed us to debug the heatmap and the privacy level. However, since we used slightly different input datasets, we disabled the option here.

- The ”Auswahl zurücksetzen”(reset selection) button resets the selection in the patient table.

Let us now take a closer look at the server application.



- ”Nummernvektor exportieren” (export number vector) provides a setup function. At the beginning of the protocol, the client sends a vector with zeros and ones. Each element of the vector represents a phone number of the service provider. The value zero or one determines whether it is requested or not. The health authority has to know the positions of their patient’s phone numbers in the numbers list. Otherwise, they would not be able to set the values correctly.

  In conclusion, we create a CSV file containing all the mobile operator’s phone numbers with this button. The phone numbers in this list contain no information about the actual owner of the number whatsoever.

  In the current version, we skip this part of the process, as we do not work on actual data.

- The function above relies on data to output. The database selection window on the right side would offer this data. At the moment, database selection is not implemented. In the current version, the data has to match internal metrics directly. It is necessary to change these metrics and recompile the program, to use another database reliably. This decision allows us to be more flexible. We will implement the database selection window once we are served data with a fixed format.

- ”Ergebnisse senden”(send results) and ”Netzwerkeinstellungen”(network settings) are deactivated. As stated above, we postponed developing a network-based service as it is not necessary to showcase the capabilities of our protocol.

- Finally, there are the ”Senderadius” (transmission radius) and the ”Privacy level” sliders. The first determines the average transmission radius of a cell tower. We are currently researching whether to omit this input or obtain it from the database. Secondly, the privacy level determines the noise added to the data in order to ensure differential privacy.

  If only a few people are in the reach of a tower, the resulting heatmap might reveal an individual’s movement. Adding noise to the result of our computation can counter this effect while still leading to useful data.

To conclude the discussion of GUI functionality, let us move to the client side again. We enter the HeatMap page by pressing ”Anfrage einlesen”(process response) in the client application.

- When selecting an item in the dropdown menu, we switch the focus of the map. In the predefined settings, we can look at the nine different counties of Austria.
- The ”Landesgrenze einzeichen” (draw county border) option draws the currently selected county border. The Austrian state provides this data.
- The ”Mobilfunkmasten einzeichnen”(drawing in cell towers) draws cell towers on the map. We used this feature to simulate data and look at tower distributions. In the current version, we only draw the cell towers in the vicinity of Vienna. We omit the rest since the resulting map file would drastically slow down the application. The towers displayed are drawn from a crowdsourced database.
- We have discussed the heatmap drawing already. The final feature left is ”Karte zurücksetzen” (reset map). We can clear up all drawings from the map by clicking this button.

2.1.3 Setup with Python

For now, we have discussed the features of our GUI implementation. In this section, however, we want to give a quick guide on setting up, modifying, and running our python implementation. We will present the git structure, the python dependencies, and the deployment pipeline.

1. The public repository of the project is located at GitHub. To get to the python GUI implementation, one has to switch to the ”develop” branch after cloning the repository.

   - `git clone https://github.com/IAIK/CoronaHeatMap.git`
   - `git checkout develop`
   - `cd pythonDemo`

   Now the project folder of the GUI should be open. We stored the precompiled version in the dist folder. To obtain it, it is necessary to install git-lfs. Git-lfs is the large file storage support from GitHub. However, it is not always up to date with all minor changes.git

2. **Dependencies:** The GUI project is written using Python 3.7. Next to Python, the following packages are necessary to run the application.

   - **PyQt5** Main framework to enable GUI programming
   - **PyQtWebEngine** Sub-framework to enable Web-views. The heatmap is internally represented as an html file. We use a Web engine to display it.
   - **Folium** is a framework used to create and manipulate JavaScript content automatically. We use it to fetch the map from OpenStreetMap and draw our heatmap.

   After installing Python with all the dependencies, we can modify and execute the application. To start the client-side run `python3 client.py` - to run the server `python3 server.py`

   We will dive more into the details of the implementation in the structure chapter.

3. **Deployment:** In order to compile our applications, we used the framework pyInstaller. It is a freely available framework for python3. Unfortunately, the folium framework does not work well together with compiling frameworks. Folium, as well as Branca, the underlying framework, use a customized Package loader. This loader is necessary to retrieve meta-data during runtime. However, we have to replace it with a system-wide loader in order for it to work with pyInstaller. To achieve this, we need to change three files from the framework. Caution! The following steps are only needed to compile the program and should be followed precisely. In the directory of python go to the `side-packages` sub directory. There you need to change 3 files in exactly the same way.

   - <PYTHON_PATH>/site-packages/folium/folium.py
   - <PYTHON_PATH>/site-packages/folium/raster_layers.py
   - <PYTHON_PATH>/site-packages/branca/element.py

   In these files you need to comment out the line where `ENV` is defined and replace it with the following:

```
#ENV = Environment(loader=PackageLoader('folium', 'templates'))
import os, sys
from jinja2 import FileSystemLoader
if getattr(sys, 'frozen', False):
        # we are running in a bundle
    templatedir = sys._MEIPASS
else:
    # we are running in a normal Python environment
    templatedir = os.path.dirname(os.path.abspath(__file__))
ENV = Environment(loader=FileSystemLoader(templatedir + '/templates'))
```

   Now that we have replaced the loader, we need to tell the compiler where to find the data files. To achieve this, we need to create a client and a server specification file. Let us look at our `client.spec` file:

```
# -*- mode: python ; coding: utf-8 -*-

block_cipher = None


a = Analysis(['client.py'],
             pathex=[],
             binaries=[],
             datas=[
                ("<PATH TO PYTHON>/site-packages/branca/*.json","branca"),
                ("<PATH TO PYTHON>/site-packages/branca/templates","templates"),
                ("/<PATH TO PYTHON>/site-packages/folium/templates","templates"),
                ],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)
exe = EXE(pyz,
      a.scripts,
      a.binaries,
      a.zipfiles,
      a.datas,
      [],
      name='client',
      debug=False,
      bootloader_ignore_signals=False,
      strip=False,
      upx=True,
      runtime_tmpdir=None,
      console=True )
```

Simply copy the existing spec file and replace <PATH_TO_PYTHON> with the actual path to your Python folder.

The datas entry can be empty for the `server.spec` file.

2.2 C++ Implementation

Before we dive into the programs' structure, we wanted to discuss the setup necessary to run the c++ implementation. To start things off, we need to clone the repository from GitHub. Then, we need to follow the README steps to download Microsoft SEAL[3] and compile the application. Execute the following commands to compile the source code:

- `bash ./install_seal.sh`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make`
- `mkdir ../bin/backend`

This procedure will clone the SEAL repository, set it to the required version, and compile it. Then, it will create a build folder where CMakeFiles, Makefiles, and Metadata will be stored. Finally, a bin directory will be created by running make from the build folder, where the compiled applications will be stored. In the final command we create a backend directory. We use this directory as a default to store all files we generate during execution.

2.2.1 Executable Files

When executing the above tasks, we create several executables. Let us now list them and briefly discuss their purpose.

- `client` This executable provides benchmarking capabilities for the client-side of the protocol. One can start it to encrypt and to decrypt. However, the application will always generate the request randomly. Both in the encrypt and in the decrypt case, we store the results in a file. We store those files in the backend folder.
- `client_gui` The client GUI executable provides an interface between our GUI and our cryptographic implementation. In addition to the encryption or decryption parameter, we also expect some options. For encryption, we must specify the following:
  - `oneHot InPath`: The path to the generated request from the GUI.
  - `Request OutPath`: The path to store the request files to.

[3] `https://github.com/microsoft/SEAL`

  - `key OutPath`: Option to store the key-material to a different directory. As we have discussed above, running the decryption requires us to remember the key used for encryption. However, we may want to store it at a different location than our request.

  For decryption, we have to provide these options:

  - `Response InPath`: The path to the response of the server. We can set this location as an input to the server GUI.
  - `Plain OutPath`: Finally, we have to tell the application where to store our computation results.

- `main` This application executes the protocol on one machine. First, we create a random request. We use this request to generate a server response. We calculate the response based on a fixed, non-linear matrix as a stand-in for our server data. Finally, we check our protocol by additionally computing the result without encryption. This procedure allows us to test the implementation both in terms of functionality and in computation time.

- `mask` In the main application, we can select execution with masking or without. However, we can only make this decision before compiling the program. Hence, the Mask executable performs the same steps as main without the functionality test, and masking turned on.

- `plain` This executable delivers a measurement of the impact of the differential privacy parameter. As an input, we expect both a path to an input file and an output file and a privacy parameter. The program takes the values from the input file and applies noise according to the privacy parameter.

- `server` Here, we provide the counterpart to the client executable. Again, this is the part solely used to test the implementation and perform benchmarking.

- `server_gui` This executable represents the interface for the server GUI and the protocol. As inputs, we can control the privacy level and the paths needed to read the request and store the results. The naming convention of the input parameters is analogous to the client GUI.

2.3 JavaScript Implementation

Important data import: As we exceeded the file size limits, we outsourced some of the essential parts [4]. On Zenodo, you should one file important to this part of the deliverable. The `data_gowalla_vienna_matrix_10000.csv` file. From the root of our attachment, one needs to put the files into this exact position.

- `project_root/Corona-heatmap-JS/input-files/data_gowalla_vienna_matrix_10000.csv` This should be the path of the data file.

[4] `https://zenodo.org/record/4293592#.X8D5RsIo_JU`

The corresponding folder with all necessary files is included in the deliverable. Its name is `corona-heatmap-JS.zip` Do not make any changes to the folder structure and execute all commands in the root folder. We need to have NodeJS and Node Package Manager installed on the executing machine. Usually, NPM is installed automatically with NodeJS. The commands needed to start the test server are the following in that exact order:

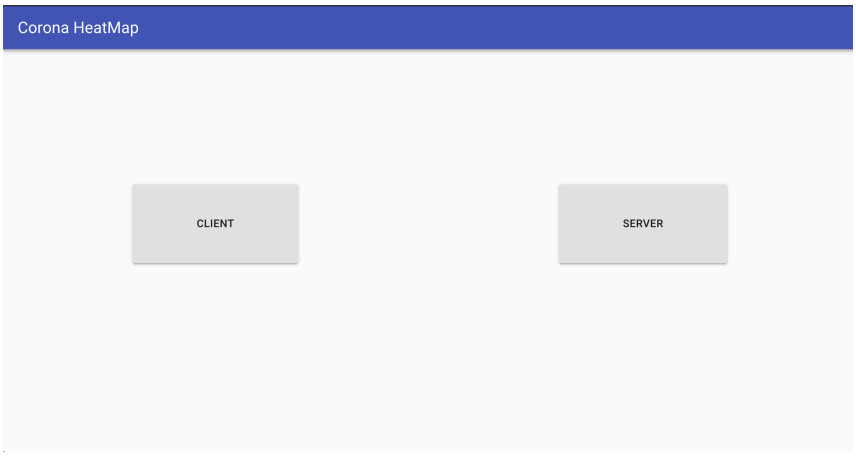
- `npm install`
- `npm run client-install`
- `npm run dev`

After that, the web app can be reached in the browser at `http://localhost:3000`.

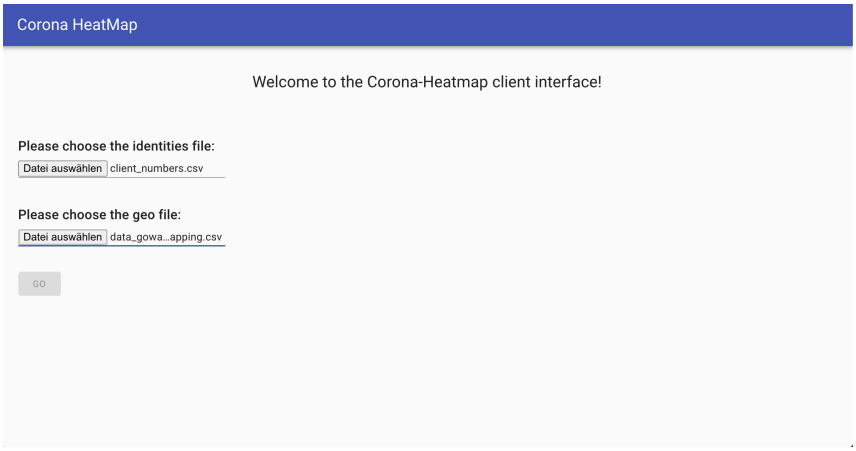
2.3.1 Demonstration

We will now walk through the process, beginning at the main page of the web app. We will have to provide some input files, which can all be found in the folder `input-files`.

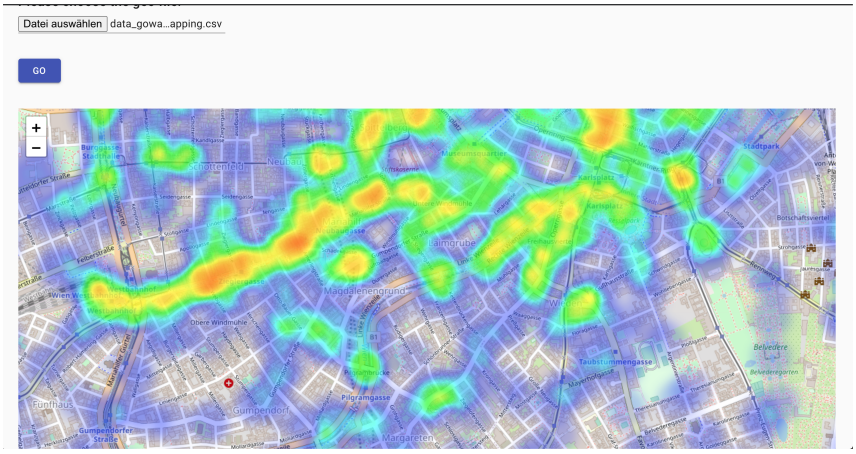
1. In the main window, we see one button for the client and one for the server. When testing the application, we recommend opening two separate browser tabs: One for the client part and one for the server part.



2. After we have clicked on the client's view, we have to provide two files to the protocol. Firstly, our identity file. Since we are only operating on the intersection of the mobile carrier's numbers and the health authority, we must first find this intersection. That is why we provide all the numbers known by the health authority in the first input field. The file is called `client_numbers.csv`. The second input field is needed for a file named `data_gowalla_locationID_mapping.csv`, which will hold the locations of the telephone poles. With this data, we will draw the map later. The `GO` button is still greyed out. We cannot press it. We first have to wait for the server to send its data. So let's move on with the server part.



3. In the server view we have to provide another two files. The first being `server_numbers.csv` which are all the numbers the mobile carrier holds. The client will find the intersection of his numbers with those numbers. The second file we have to provide is the server's matrix. In terms of the protocol this is the matrix which holds the information about how long people were at specific telephone poles. The file's name is `data_gowalla_vienna_matrix_10000.csv`. After assigning the files to the input fields we are able to click the `GO` button and we can head on to the client view again. We can press the `GO` button there too and the protocol will start to run. This can take some time. After a few minutes the heatmap will be displayed in the client window ready to be inspected.



3 Building Blocks' Structure

This chapter takes a closer look at the implementation discussed in general terms earlier. Again, we will part our discussions. First, the GUI, then the c++ implementation and finally the JavaScript part. For each part, we will describe the internal structure and significant design decisions. Then, we will briefly state the interface of the implementations. In other words, we want to clarify how to interact with our applications programmatically.

3.1 Python GUI

In the ”Setup with Python” chapter, we have already had an impression of the internal structure. We discussed dependencies as well as the deployment pipeline. We wish to take a closer look at the internal structure, directing the reader to the essential files.

Since we divide this protocol into a client and a server part, we also have two entry points in python. Those two applications are ”client.py” and ”server.py”. However, for testing purposes, we can also use the ”heatMap.py” as a standalone application. Of course, this only works with precomputed data.

As a basis for this implementation, we use object-oriented python code. The underlying GUI framework drives our class structure. Hence, we implement different GUI elements with our features. As for the heatmap itself, we use folium. Folium is a framework, which allows us to map elements in javaScript, based on the leaflet.js framework. In the ”heatMap.py,” we provide many auxiliary functions to create and edit these map elements. After each change, we save the map to an HTML file and display it using the PyQt5 WebEngineKit. This kit is a submodule of the PyQt5 framework allowing us to embed a web engine into our python GUI.

In terms of databases, we tried different approaches. For now, we work with CSV and SQLite databases. Since this is just a proof of concept, we randomly generate patients. We stored the input for this generation in the CSV files. However, we could be reading predefined patients’ data from the SQLite database. We can say the same thing for the location data. In the end, we want to be adaptable to different scenarios and willing to support different structures.

3.2 C++ Implementation

In the Introduction, we have already discussed how we can interact with our protocol’s building blocks. From testing to GUI programming, we have very different interfaces. These interfaces, however, all interact with parts of our core functionality. In this section, we want to dive into the details of this implementation.

Let us start with our dependencies. We built our protocol using the Microsoft SEAL (Simple Encrypted Arithmetic Library). This library provides us with the functionality to encrypt and decrypt homomorphically as well as performing homomorphic operations. Secondly, we make use of our shake implementation. Shake allows us to generate random field elements used in the masking process.

Let us again take a look at the client and server standalone files. In these programs, the most critical functions of the client and server are called, respectively. We can distinguish three types of methods. Framework setup, use-case specific, and I/O related.

- SEAL is very broadly applicable. Hence, we have to have a lot of initialization code. This code specifies prime sizes, key size, and other general parameters.
- Use-case specific methods setup and execute our protocol. The setup includes setting dimensions as well as reading input. The computation, on the other hand, fulfills the steps described in the paper.
- I/O related methods offer us a way to read and write input and output. Our I/O is relying on a file basis. File I/O makes it easy to communicate with other programs, such as our python GUI.

3.3 JavaScript Implementation

In this section, we are going to discuss the details of the JavaScript implementation. Since we are dealing with a Web app, there are a couple of frameworks involved in this. We will talk about the front and backend separately.
Since the core of the protocol can be generically used to build arbitrary applications similar to the Corona Heatmap, we decided to call this central part *private selective aggregation* (PSA) and derived a standalone JavaScript library for this purpose. This library is publicly available to anyone who wants to build a PSA application. The library will finally be discussed.

3.3.1 Frontend

For the user interface, we were using the popular frontend library ReactJS. React allows the developer to separate concerns nicely in the code, build reusable code modules, and eases programming overall. Since we wanted ready-to-use components like headers, buttons, inputs, and many more, we used the MaterialUI library. It offers a variety of nicely designed web components with much functionality.

3.3.2 Backend

We are using NodeJS to run an ExpressJS web server and handle the communication with the client and server from the protocol in the backend. The core of the application is built around node-seal[5], the JavsScript port of the C++ SEAL library. The interface works roughly the same, with some little caveats like performance, integer size, and object allocation. In the folder structure, there is a server and a client part. This may seem confusing at first. However, this has nothing to do with the client and server from the protocol. It is just a separation of the frontend (client) and the backend (server). All the code related to the protocol is in the `client/src/util`. There are the files `clientConnection.js` and `serverConnection.js`, which serve as the main code files for client and server from the protocol. All the code related to the protocol's mathematical operations is located in `client/src/util/MatMul.js`.

3.3.3 PSA Library

To build arbitrary PSA applications, we developed a dedicated JavaScript library with an easy to use interface. The library is included in the deliverable. The corresponding zip file is called `PSA.zip` The library can be parametrized to fit it even to complex PSA applications.
The library is available through NPM[6]. It can be installed in any node project by just executing `npm i psa-lib` in the project's root. The API was thoroughly documented in the corresponding GitHub page[7].

[5] `https://github.com/morfix-io/node-seal`
[6] `https://www.npmjs.com/package/psa-lib`
[7] `https://github.com/Safe-DEED/PSA`

4 Conclusion

In this deliverable, we have presented implementations of a novel privacy-preserving protocol. Its applications are broad and not yet fully explored. We already have ideas for further business use-cases for the protocol. The next big step is to integrate the JavaScript Library to the WP4's Data Valuation Component Questionnaire. Since the protocol was designed for scalability, we do not envision any substantial performance issues for this use-case even though it will be a web-application.