Grant Agreement Number: 825225

Safe-DEED

www.safe-deed.eu

D5.12 - Implementation of cryptographic building blocks and specialized protocols v3/3

Deliverable number	*D5.12*
Dissemination level	*Public*
Delivery data	*due 30.11.2021*
Status	*Final*
Authors	*Lukas Helminger, Stefan Lontschar, 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
08.10.2021	Fabian Schmid	SLTBP Section	0.1
19.10.2021	Stefan Lontschar	PSI Section	0.2
21.10.2021	Fabian Schmid	PSA Section	0.3
25.10.2021	Stefan Lontschar	PSI Benchmarks	0.4
02.11.2021	Stefan Lontschar	PSI library	0.5
03.11.2021	Lukas Helminger	Intro, Conclusion	0.6
04.11.2021	Lukas Helminger	Executive summary	0.7
15.11.2021	Abdel Aziz Taha (RSA)	Safe-DEED Internal Review	0.8
23.11.2021	Lukas Helminger	Final Editing	1.0

Executive Summary

This deliverable - D5.12 Implementation of cryptographic building blocks and specialized protocols v3 - is the final software result of Safe-DEED's WP5 in the form of a demonstrator. This report provides an update and an overview of the integration of three major secure computation components - a private set intersection (PSI) library, a private, selective aggregation library, and a secure lead-time-based pricing program - deployed in Safe-DEED.

This demonstrator consists of a new PSI library developed together with the EU Horizon 2020 project TRUSTS. The previous version lacked scalability for enterprise-scale data sets. The new version is business-friendly and focuses on performance and reliability. In this document, we report on extensive real-world performance experiments that show scalability - around 10x times faster than the old solution.

The task for the other two software components was to integrate them into the existing business environment and processes. The Private Selective Aggregation library was successfully integrated into WP4's Data Valuation Component. The web-based solution allows users of the Data Valuation Component to protect their answers to the questionnaire with homomorphic encryption without compromising the accuracy of the result.

For the Secure Lead-Time-Based Pricing - together with the use case partner in WP7 - we bridged the gap between WP5's command-line program to WP7 graphic user interface. This report describes the integration process as well as a Docker version to run the program as a stand-alone application.

Table of Contents

List of Figures

Abbreviations

ATP	Available to Promise
DVC	Data Valuation Component
FRESCO	Framework for Efficient and Secure Computation
GUI	Graphical User Interface
SLTBP	Secure Lead-Time Based Pricing
JRE	Java Runtime Environment
JDK	Java Development Kit
JSON	JavaScript Object Notation
PSA	Private Selective Aggregation
PSI	Private Set Intersection
TLS	Transport Layer Security

1 Introduction

The purpose of this deliverable is to report on the implementation and integration of cryptographic protocols deployed in Safe-DEED. It provides a demonstrator consisting of three independent software libraries and this corresponding report. This report is divided into three sections, each describing one of the software components.

1.1 Private Set Intersection (PSI)

We presented our Private Set Intersection (PSI) library in the previous deliverable D5.4. This first version of the library was using a Java PSI library, with the core of the cryptographic operations being executed by a component written in C++. While this previous implementation was sufficient for the purpose of building a small library, further integration work has shown that the performance and stability of the Java PSI library was lacking for enterprise-scale data sets. To combat these issues, a new version of the PSI functionality was developed from scratch in cooperation with the EU Horizon 2020 project TRUSTS[1]. The focus was put on performance and reliability. The new version was successfully integrated into the WP6 demonstrator[2]. In addition, the PSI library was added to the EUHubs4Data catalog[3].

1.2 Private Selective Aggregation (PSA)

We presented our Private Selective Aggregation (PSA) library[4] in the previous deliverables. D5.8 described the theoretical foundations, whereas D5.9 discussed the first implementations efforts. D5.11 offers a web-based solution and a detailed description of how it can be used in Eurecat's Data Valuation Component (DVC). This deliverable focuses on the integration of the library as a software component into the DVC.

1.3 Secure Lead-Time Based Pricing (SLTBP)

This section focuses on the integration of the Secure Lead-Time-based pricing (SLTBP) demonstrator. We have discussed the initial program and its toolchain in deliverable D5.4. Then, we have presented the follow-up developments in deliverable D5.11. Now, we want to give an overview of the integration of the project and its usage. We developed the SLTBP java application as a cryptographic compound, meaning that we optimized interaction with the program for development. We provide a folder structure and appropriate "make targets" to test the different algorithms. The use case partner worked on a graphic user interface (GUI) to bridge the gap between our program and end-users. In this deliverable, we will walk through the necessary steps along the way and the improvements we made to facilitate integration. Finally, we want to discuss the remaining challenges, first and foremost the performance issues.

[1] https://www.trusts-data.eu/
[2] https://demo.safe-deed.eu/
[3] https://euhubs4data.eu/services/know-psittacus-privacy-enhancing-technology-for-data-sharing/
[4] https://github.com/Safe-DEED/PSA

2 PSI Library

2.1 Technical Overview of the PSI Library

In contrast to the first library ($v1$), we build the second PSI library ($v2$) in *Rust*. Rust is a modern programming language with focus on performance and reliability. One of the main important features of Rust is its focus on memory safety, with its *borrow checker* component that ensures memory safety and removes large classes of common, often security-critical errors such as use-after-free errors and buffer overflow erros.

Used private set intersection protocol. The previous library $v1$ used the PSI protocols developed for private mobile contact discovery of [2]. While these protocols can perform well for large set sizes (up to multiple million elements), their relative internal complexity also makes some aspects of the implementation more complex. Furthermore, if both datasets are relatively small (less than one million elements each), the simpler protocol in [1], which is based on a variant of Diffie-Hellman key agreement, can perform nearly as well computationally while allowing for reduced communication overhead compared to the protocols in [2]. The basic nature of the [1] protocol still follows the high-level description outlined in deliverable D5.4, Figure 1. In our implementation, we also apply some of the optimizations of [2] to the protocol of [1], namely the use of a cuckoo-filter with small false positive probability and cuckoo filter compression.

Implementation details. We additionally protect the communication channel between the two parties using a TLS connection. For this, we use the `rustls`[5] library, an implementation of TLS in the Rust programming language. Our implementation allows for both, self-signed certificates, as well as traditional public-key infrastructure. We use TLS version 1.3 per default.

2.2 Performance Evaluation

To objectively quantify the improvements of the new PSI library $_v2$, we repeat and extend the performance benchmarks previously carried out as part of deliverable D5.4.

We benchmark the previous library $v1$ as well as the new improved $v2$ one in multiple settings:

- A setting where both parties are in the same network (e.g., the same datacenter, Frankfurt in our case)
- Two settings where the two parties are in two separate geographical areas:
  - Frankfurt-Paris
  - Frankfurt-Ohio

The benchmarks were executed on Amazon Web Services (AWS) EC2 servers with both parties running an Ubuntu Server 20.04 LTS image on a `c5.xlarge` configuration.

[5] `https://github.com/rustls/rustls`

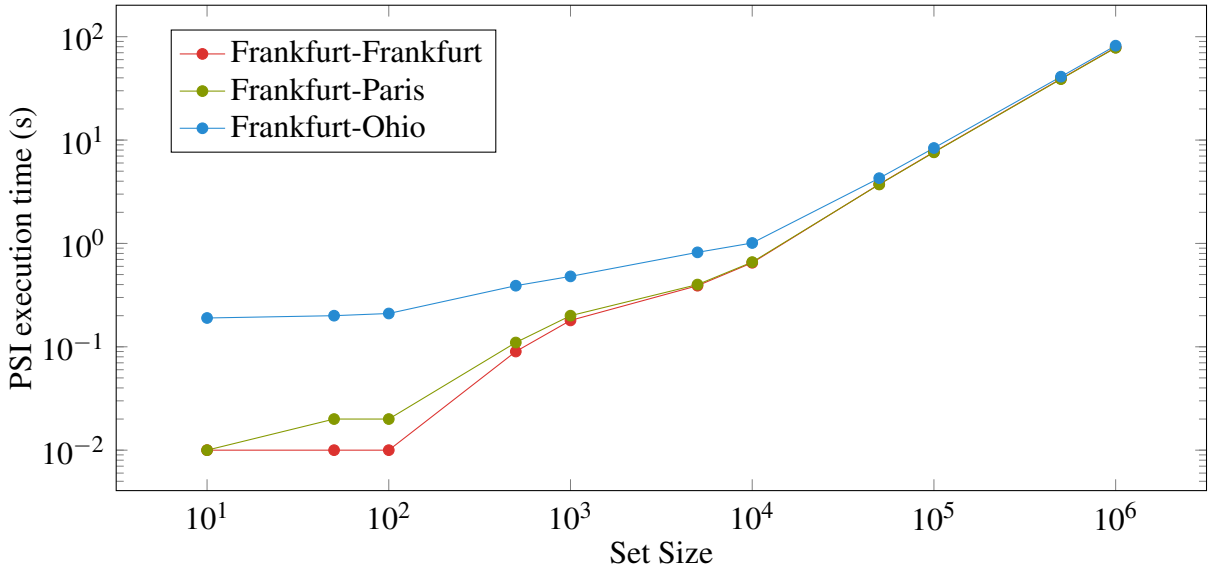

Figure 1: Runtime of the PSI library for symmetric sets (# of client items = # of server items) in different networking configurations.

We first show the runtime of the new PSI library in Figure 1 and the network communication of the different parties in Table 1. We can see that for small set sizes, the additional latency in the Frankfurt-Ohio scenario leads to an increased runtime, however, this small additional latency is insignificant for larger set sizes and the three different networking scenarios are practically identical in terms of runtime.

We compare to the previous PSI library prototype *v*1 included in D5.4. The runtime of the old library *v*1 can be seen in Figure 2. We only benchmark small set sizes due to the memory usage constraints of *v*1. The new PSI library *v*2 is usually about 10x times faster than the old one, e.g., if both parties hold a set of 5 000 items, *v*2 takes about 0.39 seconds, while *v*1 took 4.54s in the Frankfurt-Frankfurt scenario. We can also see that the old library *v*1 did not optimize the networking round-trips in Figure 2, as there is a large difference between the different networking scenarios, e.g., for 5 000 items, the runtime in the Frankfurt-Frankfurt scenario is 4.54s, while the runtime in the Frankfurt-Ohio scenario rises to 58.86s.

The underlying PSI protocol used in the PSI library *v*2 is suited to the scenario of imbalanced set sizes[6], where larger server set sizes are more beneficial for the protocol. This can be seen in Table 2, where we repeat the same scenario twice with exchanged sets. Here, the run where the server has the larger set size is faster, and additionally reduces the amount of data transferred by a large amount.

[6]The server set size is larger than the client set size by order of magnitudes.

Set size	Server (KB)	Client (KB)
10	1.60	1.99
50	4.38	6.05
100	7.81	11.08
500	34.07	50.14
1 000	66.07	98.14
5 000	322.39	482.62
10 000	650.58	971.01
50 000	3 213.71	4 815.70
100 000	6 417.66	9 621.60
500 000	32 048.94	48 068.52
1 000 000	64 087.98	96 127.09

Table 1: Communication sent by parties in the PSI library for symmetric sets (# of client items = # of server items) in all networking configurations.

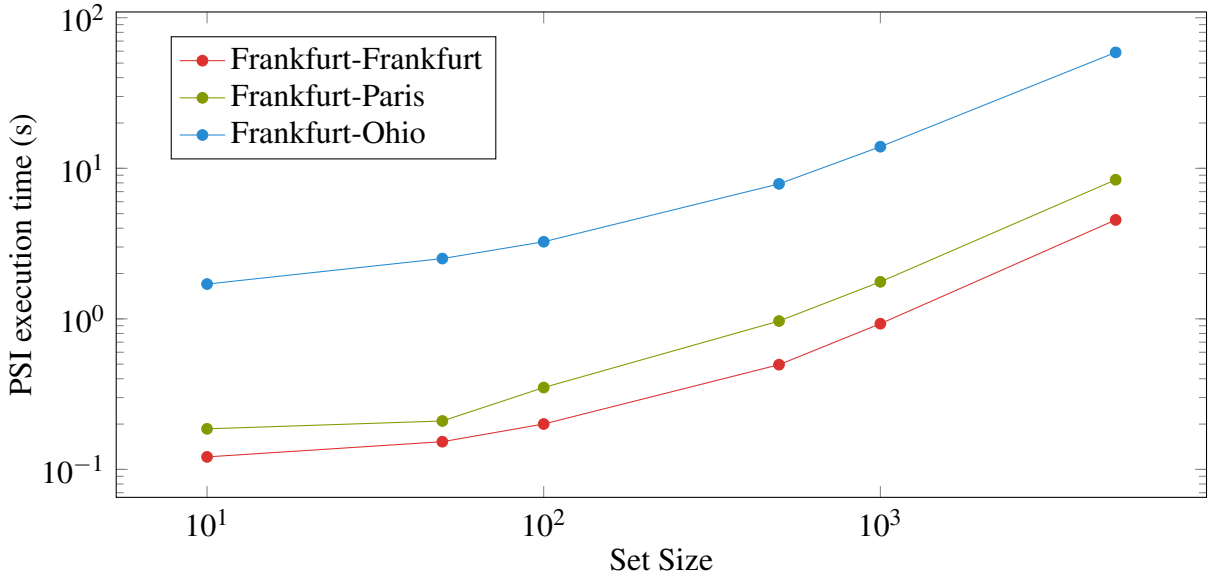

Figure 2: Runtime of the old PSI library from D5.4 for symmetric sets (# of client items = # of server items) in different networking configurations.

N_s	N_c	Runtime (s)	Server (KB)	Client (KB)
1 000	100 000	5.53	9 613.78	6 409.84
100 000	1 000	2.84	105.97	73.90
10	1 000 000	54.64	96 118.15	64 079.04
1 000 000	10	27.29	10.93	10.54

Table 2: Runtime and communication in the PSI library for asymmetric set sizes (server set size N_s, client set size N_c) in the Frankfurt-Frankfurt networking scenario.

2.3 Running the Library

The PSI library can be found in the accompanying `psittacus-bin.zip`. The information below is repeated in the included `Readme.md` and running the binary with a `--help` argument also prints a detailed usage message.

PSIttacus - Binary

This is a description on how to use the compiled version of `PSIttacus`. With the compiled version it is possible to only write a configuration file in Yaml format, supply that to the binary to start the PSI process. If a config file cannot be provided, all required attributes can also be specified via a CLI. Then, two parties can securely calculate a set intersection without revealing any data to the other party. Depending on the configuration the intersection can be revealed to either or both parties as a result of the calculation.

The following examples currently need to be run from the source of this repository to work. All paths specified should be relative to the calling directory.

Example `config` structure

```
---

log_level:  Error

network:
    address:    127.0.0.1:7878

psi:
    is_server:          true
    data_file:          data/micro.csv
    result_file:        intersection.csv
    server_to_client_full_data: false
    client_to_server_full_data: true
```

Configuration Attributes

- `log_level`: Defines the level of information to be printed during the process. Possible values are: `Trace`, `Debug`, `Info`, `Warn` and `Error`. Defaults to `Error` if not supplied.
- `network`: Section for settings related to networking
  - `address`: Defines the socket as ip_address:port the participant should connect to. Server side creates a listener on this socket address and client side will connect to it.
  - `tls`: Optional section for settings related to TLS
    - `allow_pki`: Defines if Mozilla PKI should be trusted and used. Either this needs to be defined and `true` or `trusted_certificates` need to be defined for a client to be able to authenticate. Only valid for the client and defaults to `false` if not defined.

    - `trusted_certificates`: Should point to a .PEM file containing a trusted certificate chain used for authentication. Required for Server and Client if authentication is mandatory.
    - `certificate_chain`: Points to a .PEM file containing the certificate chain to be used during authentication process. Always required for the Server and only required for the client if authentication will be done.
    - `key`: Points to a .PEM file containing the key for the personal certificate that is contained in the chain defined in the file of `certificate_chain`. Always required for the Server and only required for the client if authentication will be done.
    - `require_client_auth`: Defines if connecting clients need to authenticate in order to connect to the server.

- `psi`: Settings related to the PSI protocol
  - `is_server`: Defines which side the party is going to pose as in the PSI process.
  - `batch_size`: The amount of messages that are being batched and sent together in the internal OPRF calculation process. The default of 1024 should suffice for most uses. Only required for the Client.
  - `data_file`: Path to the data_file that should be used for set intersection calculation, relative or absolute.
  - `result_file`: Path where the resulting intersection should be saved, relative or absolute. Always required for the Client and also for the Server if `client_to_server_full_data` is active.
  - `psi_column`: Defines which column-id of the provided data-set should be intersected with the other parties data. Defaults to `id` if not provided.
  - `server_to_client_full_data`: Defines if the result of the PSI process and all corresponding columns of each intersecting row should be sent from `Server` to `Client` after the process. This field needs to be set identically for both participants for the process to work and will be checked at the start.
  - `client_to_server_full_data`: Defines if the result of the PSI process and all corresponding columns of each intersecting row should be sent from `Client` to `Server` after the process. If this is set to `true` a `result_file` path needs to be specified for the server. This field needs to be set identically for both participants for the process to work and will be checked at the start.

CLI Attributes All leaf attributes of the config file can also be provided to the binary via the command line. Either complete or as single overrides over the config file.

General Usage

1. Prepare data to be processed in CSV format with an ID column (named `id` by default, can be specified via the `psi_column` configuration option). The intersection will happen on

the ID column, with the other columns being associated information that can be optionally exchanged after the intersection (which can be specified via `server_to_client_full_data` and `client_to_server_full_data` config values).

2. Create config.yml file to be used by the binary according to the description in the section 'Example config structure' or specify required attributes via CLI as described in the following 'How to call the binary' section.

3. Run the binary as specified in the following section.

The binary needs to be configured via the CLI or a dedicated config file. Some examples can be found below

Using a config file:

```
./psittacus --cfg config.yml
```

Using a config file and overrides:

```
./psittacus --cfg config.yml --log-level Trace --psi-column v0
```

Server, from CLI (with TLS params):

```
./psittacus --log Debug \
--server true \
--data data/datafile.csv \
--result data/result.csv \
--client-to-server true \
--variant Balanced \
--col v0 \
--address 10.0.0.128:7878 \
--trusted certificates/certificate.pem \
--key certificates/key.pem \
--certificate certificates/chain.pem \
--auth false
```

Client, from CLI (with TLS params, no client authentication):

```
./psittacus --server false \
--data data/micro.csv \
--client-to-server true \
--result data/result.csv \
--trusted certificates/certificate.pem \
-b 512 \
-a 127.0.0.1:7878
```

Client, from CLI (no TLS, minimal):

```
./psittacus --server false \
--data data/micro.csv \
--result data/result.csv \
-b 512 \
-a 127.0.0.1:7878
```

How to run the provided library

The provided library includes both configuration files, example data sets and TLS certificates to allow executing a fully self-contained example.

Start the server:

```
./psittacus-linux-amd64-0.3.2 --cfg tls_server.yml
```

Start the client:

```
./psittacus-linux-amd64-0.3.2 --cfg tls_client.yml
```

Expected output On success both parties write a confirmation to the command line depending on the value of `log_level`.

The client also writes the received intersection into a file (as specified in the config). Depending on the configuration (`X_to_X_full_data` values in the config) either party also saves the full intersection data in a corresponding `intersection.csv` file.

3 PSA Library

3.1 PSA Integration Caveats

The development and internals of the PSA library were desired in previous deliverables, see Section 1.2 for details. Here, we want to summarize the experience gathered in the integration process.

Configuration Options Microsoft seal and with it node-seal are very context-heavy frameworks. Developers can leverage homomorphic encryption (D5.1) in a variety of settings with different security assumptions. Hence, the myriad of customization options. We can fine-tune the seal libraries to meet exactly our needs. However, for everything to work as expected, we need some additional synchronization effort. The Server and the Client must agree on the same security parameters in our PSA library before beginning the protocol.

Software Architecture Due to the nature of the implemented algorithm, the library has to run on two devices. In the risk assessment use case we presented in deliverable D5.11, both the Server and the Client were using a browser as depicted in Figure 3. This setting's main advantage is the ease of access. Both parties only need to enter the website and upload their respective data files. The obvious disadvantage is that the vector-matrix multiplication takes place in the browser of the Server party. Depending on the machine in use, this might have a significant performance penalty.

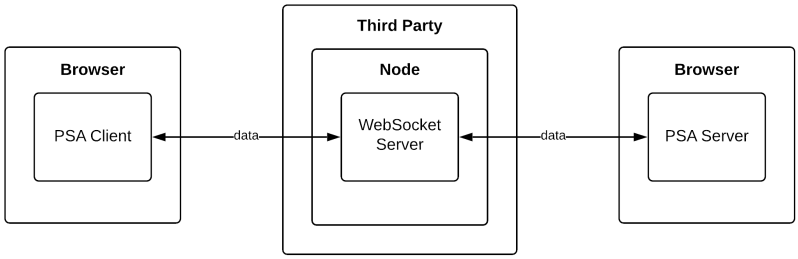

Figure 3: Browser to Browser PSA Architecture

In the case of the DVC by Eurecat, we use the library in a slightly different architecture. Here, the actual web server runs the Server part of the computation as depicted in Figure 4. With this setup, we reflect the structure we used to benchmark the library in D5.11. However, in this scenario, other challenges arise. The PSA library was designed with ease of implementation in mind. With both the front-end and the back-end being in JavaScript, we hoped that small node-based applications would be easy to build. As this remains true, integrating the PSA library to existing non-node-based back-ends proved more challenging.

In the existing architecture, the back-end consisted of a python flask server. In the current solution, this flask server has to communicate to a separate express server dedicated to the PSA calls of the back-end. While having a different server for different components is fine, it raises the question of performance. In such an architecture, it might be most beneficial to

re-implement the server part of the PSA library in C++ and perform those computations on a dedicated server. Finally, we can see that computationally intense applications such as those using homomorphic encryption need to be fine-tuned on parameter and architectural levels.

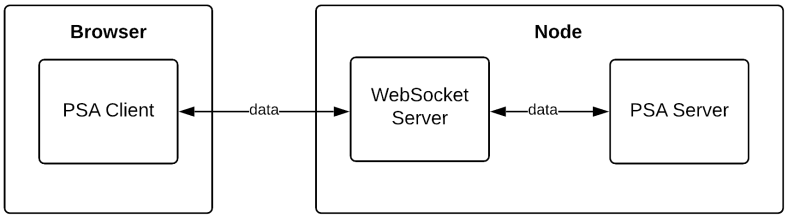

Figure 4: Browser to Node PSA Architecture

Responsiveness JavaScript and, therefore, the PSA library run single-threaded. It is essential to know that the computations are blocking. In other words, if the server-side computation is executed in a browser environment, the interface will freeze. This lack of responsiveness does not indicate an error. For this reason, we added a spinning wheel to our risk analysis implementation. However, we suggest using the native C++ version on the server-side if possible.

4 Secure Lead-Time-Based Pricing

4.1 Running the Demonstrator

We will quickly repeat the methods on how to run the demonstrator as a standalone application. The code to the java command line demonstrator is publicly available at GitHub[7].

4.1.1 Linux

Using the **Linux** platform, we need `make` and a java runtime environment `jre`. We can then call the different pricing protocols with the following calls.

```
make linear
make convex
make concave
make bucket
```

These calls will execute 3 parties using the predefined settings in the respective demo directories. In each of those demo runs there is one vendor as server1 and two vendees as server2 and server3. They perform the pricing evaluation for three individual products with different price offers and delivery times. The only difference between the folders `linear`, `convex`, `concave`, `bucket` is the underlying pricing algorithm used. By reading one of the servers `accepted_orders.json` in one of the directories we can see which `SalesPositions` succeeded. The input files were selected in such a way that each algorithm accepts different products. The result files are provided in the repository. The runtime was between 18 and 27 minutes depending on the protocol. These values should be taken with a grain of salt, and serve as reference points. When measuring the timing of the demo setups runs we used the following CPU: Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz

Applying changes to the library requires a java development kit (`jdk`) and `maven` to build the code. The following command installs all necessary dependencies on Ubuntu.

```
sudo apt-get update && sudo apt-get install -y \
  openjdk-11-jdk \
  maven \
  make
```

The next calls build the code and move it to the correct locations in the demo sub-folders.

```
make install
make move
```

4.1.2 Docker

Platform independence is achieved by using our provided Dockerfile. Using a docker command line interface, we can run our container as follows.

```
docker build -t sltbp .
docker run -i sltbp
```

[7] https://github.com/Safe-DEED/SLTBP

The first command will create a docker container from an Ubuntu image. This container will have the above mentioned dependencies installed. It will then copy the current directory (i.e., the root of SLTBP) to the container and run `make install` and `make move`. This will assure, that any prior changes are reflected in the demo directories of the docker container.

The second instruction will run an interactive bash in the SLTBP directory of the docker container. There we can run the demo setups with the above mentioned make targets.

4.2 Integration of the Code

Integration with Maven The integration procedure of SLTBP depends on the starting point. The easiest approach would be using java and maven. In this scenario we only need to run `mvn clean install` in the root directory. This will install SLTBP in our local maven repository. Then, we include SLTBP as a dependency in our pom.xml as follows:

```
<dependencies>
    <dependency>
        <groupId>iaik</groupId>
        <artifactId>root-mpc</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    ...
</dependencies>
```

We can then call the static function `secureLeadTimeBasedPriceFinder()` to start the SLTBP. This is equivalent to executing the demo.jar file in the current directory.

Running SLTBP as a seperate process The above mentioned approach, however, is not compatible with different platforms or build tools. We therefore decided to rely on JSON files as inputs and interfaces to our program. In this way we can set up all the parameters with the input files and execute the SLTBP demonstrator as a standalone java application. We will name and describe the 3 input files and the output file of the demonstrator.

- `ATPUnits.json` This file contains the input to the MPC protocol as described in previous deliverables. A single ATPUnit mus contain of a Date, Price, Amount and Sales Position. The last must be unique in the list and shared with all other participants. An overview is given in table 3
- `NetworkConfig.json` Here we have to provide information on the entire network of participants. One party is described with an ID, IP, Port and a boolean signaling if the object describes oneself. Again summarized in table 4
- `MPCSettings.json` With this deliverable, we decided to add this configuration file. In D5.11 we stated in chapter 3.2.2 how the most important MPC settings could be altered. As this process required recompiling the project, we now provide file based settings. The updated configuration options are listed in table 5.
- `accepted_order.json` This file is used as an output for the protocol. The client stores its succeeded deals, the server stores all succeeded deals. In case all Sales Positions fail,

the key-value pair `deal:failed` is stored. The structure of the JSON objects is the same as in ATPUnits.json. The only difference is that the price is given as the aggregated price of this order.

Type	Name	Description
`String`	`date`	Setting ordered lead time for client and standard lead time for server
`String`	`amount`	Setting amount of units ordered and total, for client and server respectively
`String`	`price`	Setting the offered price for a single unit
`String`	`Sales Position`	Unique identifier connecting the unit to a real product

Table 3: A single ATPUnit

Type	Name	Description
`String`	`id`	Server has to have `id=1`. The remaining ids are ascending
`String`	`ip`	IP address of the player with that id
`String`	`port`	The port of the player with that id
`Boolean`	`myID`	A boolean value indicating whether this object describes myself

Table 4: A Network Configuration Object

Type	Name	Description
`String`	`evaluationProtocol`	Defines the pricing protocol to run, if benchmark is disabled
`String`	`preprocessing`	Defines source of multiplication triples. Must be set to MASCOT for secure computations
`String`	`otProtocol`	Must be set to Naor, to enable the secure Naor Pinkas OT protocol
`String`	`evaluationStrategy`	Defines evaluation strategy for native protocol blocks inside FRESCO
`String`	`maxBitLength`	Defines the maximum bit length of secret shared values
`String`	`modBitLength`	Defines the bit length of the modulus
`Boolean`	`benchmarking`	Runs all pricing protocols after one another and tracks their time and network usage
`Boolean`	`debug`	Runs all pricing protocols in plain to check results

Table 5: Contents of MPCSettings.json

With the GUI provided by Infineon we can visualize the steps in the process mentioned above. First, each party enters their input as shown in Figure 5. Then, the parties wait while the MPC

computation is performed (Figure 6). Finally, the parties receive the result as described above and shown in Figure 7.

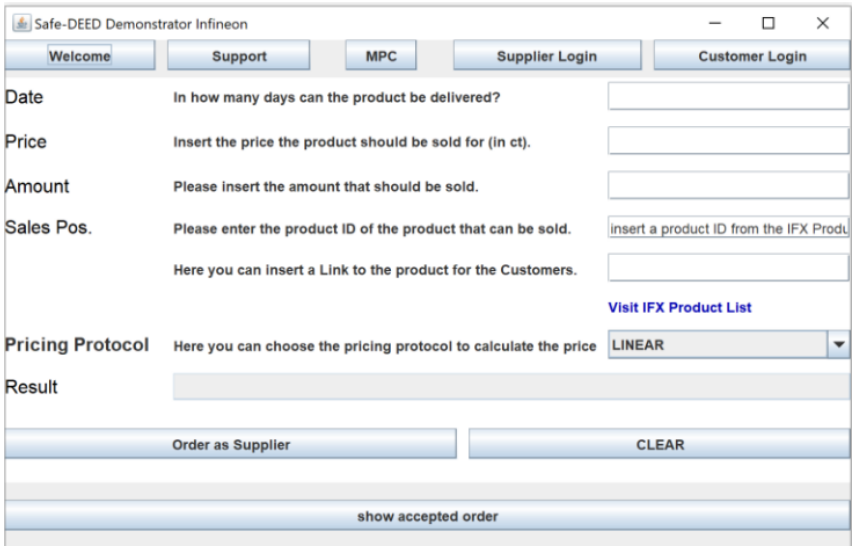

Figure 5: Interface to create `ATPUnits.json`

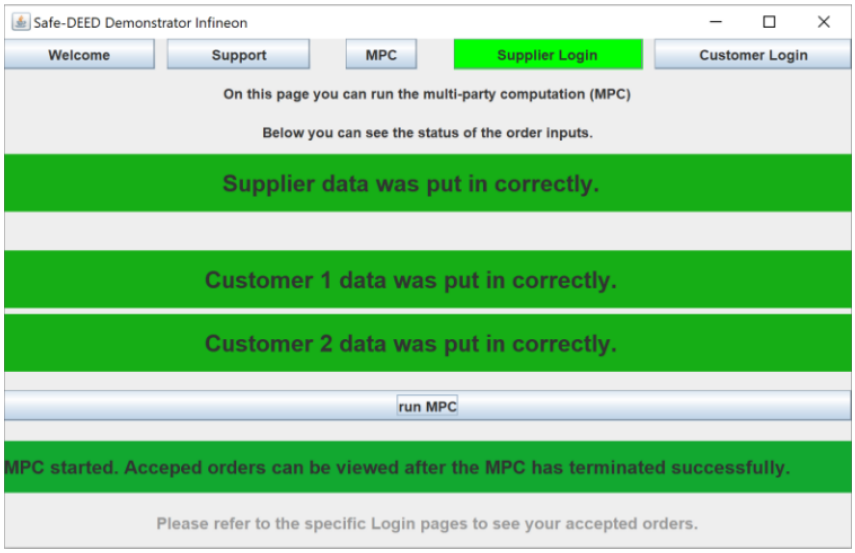

Figure 6: Interface view during MPC computation

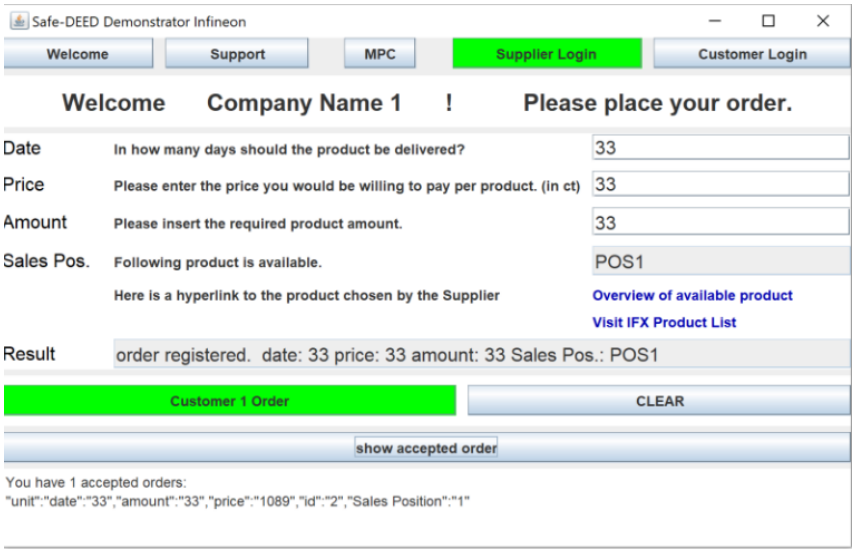

Figure 7: Interface showing result of `accepted_orders.json`

4.3 Recent Changes

For the final deliverable, we mainly focused on support for the integration of our demonstrator. However, we found inconveniences, bugs, and unnecessarily complex constructions during the process, which we then refactored. In the following section, we want to overview the changes to the code that happened since the previous deliverable D5.11.

Silent participant As a measure of convenience, we added the option of a silent participant. Our use case partner specifically requested that not all participants should have to order all listed sales positions. This change will only trigger our aggregation protocol once there is an overlap in the requests of customers. Each participant still needs to have a JSON object for each sales position. However, if the party wants to abstain from ordering, they need to set the date of those sales positions to zero. They will then participate in the MPC computation for that unit, yet they will not provide any input.

Opening bug We noticed that some of our protocols would not correctly open the final result of the computation. This problem led to the creation of a common OpenProtocol. This protocol opens the ATPUnits with the final prices and prepares a data structure to create the JSON output file.

Refactoring With the last deliverable, we have already changed from two different applications to a single one. In the beginning, we started with a client and a server package. We moved the computation to the Application package and made the user selection dependant on the input files. We made the necessary cleanup with this deliverable by moving needed helper functions to either the Util or the Application package and deleted the Client and the Host package entirely. This refactoring also streamlined the structure of our pom.xml files.

5 Conclusion

The technical solutions presented here will directly contribute to the adoption of the two main principles of the GDPR. It will enable data sharing while protecting individuals' privacy. Thus, it will help position European enterprises as market leaders in responsible data-driven businesses.

We now summarize three high-level conclusions we draw from developing and integrating these three privacy-preserving software libraries.

- Privacy-preserving protocols should be more heavily encouraged or even demanded by regulators, e.g., through Article 25 (Privacy by Design) GDPR or the GDPR codes of conduct requiring privacy-preserving algorithms in specific settings.
- The community should not only focus on efficiency but also on developing more business-friendly privacy-preserving solutions, i.e., one with a high technology readiness level and simple integration into existing systems.
- The integration of privacy-preserving software libraries still needs privacy engineers experts.

6 References

[1] Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *SCN*, volume 6280 of *Lecture Notes in Computer Science*, pages 418–435. Springer, 2010.

[2] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *USENIX Security Symposium*, pages 1447–1464. USENIX Association, 2019.