Grant Agreement Number: 825225

Safe-DEED

www.safe-deed.eu

Protocols for Privacy-Preserving Data Analytics and Secure Lead-Time Based Pricing v2/2

Deliverable number	*D5.11*
Dissemination level	*Public*
Delivery data	*due 31.05.2021*
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
07.05.2021	Alexander Grass, Lukas Helminger, Fabian Schmid	First Draft (for internal review)	0.1
12.05.2020	Tobias Leander Welling, Fabian Gassner, Trung Nguyen, Jan-Philipp Erdmann	Internal Review	0.2
17.05.2020	Mihnea Tufis	Internal Review	0.3
26.05.2020	Alexander Grass, Lukas Helminger, Fabian Schmid	Final Version (incorporated reviews)	1.0

Executive Summary

This deliverable D5.11 - Protocols for Privacy-Preserving Data Analytics and Secure Lead-Time Based Pricing v2/2 - is an update of D5.4. It is the final outcome of task T5.2 and of great importance to achieve milestone MS8. This task was concerned to develop suitable protocols for the Safe-DEED use-cases based on protocols developed and knowledge gained in task T5.1. The protocols from this deliverable will provide the functionality required by the work packages WP4, WP6, and WP7. The deliverable consists of a demonstrator and this report. The demonstrator can be divided into two components.

The Private Selective Aggregation (PSA) library is based on earlier research efforts (see D5.8 and D5.9). This lightweight library shows that privacy-preserving computations can even be done in the browser. The PSA library is a prime example of privacy by design. It offers a chance for data minimization. The plan is to integrate the PSA into Safe-DEED's data valuation component developed in WP4.

The secure lead-time based price protocol (SLTBP) introduced in deliverable D5.4 was developed further. The pricing function was jointly developed by multi-party computation (MPC) and lead-time pricing experts. In this way, it was guaranteed that on the one side, the pricing functions reflect the real-world and, on the other side, are MPC-friendly, i.e., (relative) efficiently computable. The report contains the results of extensive benchmarking tests in various settings to show the scalability of the SLTBP protocol.

Table of Contents

List of Figures

Abbreviations

API	Application Programming Interface
ATP	Available to Promise
DVC	Data Valuation Component
FRESCO	Framework for Efficient and Secure Computation
LTBP	Lead-Time Based Pricing
MASCOT	Faster Malicious Arithmetic Secure Computation with Oblivious Transfer
MPC	Multi-Party Computation
NPM	Node Package Manager
PSA	Private Selective Aggregation
SLTBP	Secure Lead-Time Based Pricing
SPDZ	MPC Protocol of **S**mart, **P**astro, **D**amgard, and **Z**akarias
SPDZ2k	SPDZ mod 2^k
TLS	Transport Layer Security
WAN	Wide Area Network

1 Introduction

The purpose of this deliverable is to report on the cryptographic protocols developed for privacy-preserving data analytics and secure lead-time based pricing. It provides a demonstrator consisting of two independent software libraries. The PSA library is a versatile privacy-enhancing protocol. It is planned to integrate the library into the Data Valuation Component (DVC) in WP4. The DVC use is best shown in the Safe DEED's WP6 demonstrator. The SLTBP protocol was developed to improve price and delivery time in the semiconductor business. This report aims to make the software libraries more accessible by describing their functionalities, providing information on design choices as well as performance, and how they benefit Safe-DEED objectives.

1.1 Related Documents

Deliverable D5.11 is the updated version of D5.4. One part of D5.11 is based on the protocols of deliverable D5.8 and their implementations in D5.9. The other part is highly influenced by the pricing algorithms described in deliverable D7.5 and D7.10 from WP7.

1.2 Roadmap

This report is divided into two main parts. First, in Section 2, we describe the updates in the area of privacy-preserving data analytics. More concretely, we report on the progress of the PSA library, scalability, and how we plan to integrate it into WP4's data valuation component. Secondly, in Section 3, we give an extensive overview of the updated SLTBP protocol, including rigorous benchmarks. In Section 4, we sum up our findings and examine how the protocols could be used in the future.

2 Privacy-Preserving Data Analytics

This section describes the progress of the PSA protocol. We developed a fully operating Web-library, a prerequisite for integration into Safe-DEED data analytics components. The benchmarks show that although there is a significant overhead resulting from the privacy-enhancing technologies, there are several possible use-cases. To see better how PSA can be applied outside from Safe-DEED, we constructed an additional use case in the finance sector.

All of the following subsections are based on Alexander's Grass master's thesis [1]. It was conducted while Alexander Grass was working in Safe-DEED and also co-supervised by Safe-DEED researchers. The thesis was successfully defended on the 29th of April and will be published soon by the TU Graz[1]. For a very detailed explanation of PSA, its use, and the technical details, we refer to the master thesis.

2.1 Usage

The library is publicly available and accessible by the Node Package Manager (NPM)[2]. It can be used in any node-powered environment that supports package management by any typical managers like NPM or yarn. This includes React, Angular, Express.js applications, and many more. The PSA library is installed by running `npm i psa-lib` from the command line so that the bundle gets registered in the corresponding package.json file. The library's interface is intuitive and user-friendly. The source code, together with API documentation, is available in the official GitHub repository[3].

In the Readme, we discuss the library's installation procedure and give a precise description of implementation details. If a user already knows which application to build on top of the PSA library, all relevant information can be found in the repository. However, if a user is unsure which problems this library is most suited to solve, the following section gives an overview of different possibilities. We summarize the essential information in respect of developing applications with the library:

1. Client and Server agree on a set of parameters in advance
2. Client and Server instantiate the library on their end (e.g., `import PSA from 'psa-lib'`)
3. Client and Server create a corresponding context with the parameters agreed on
4. Client passes his data to the library for encryption
5. Client transmits resulting encrypted vector to the Server
6. Server passes his matrix and the Client's encrypted vector to the library and computes the result vector
7. Server sends result vector back to the Client
8. Client decrypts the resulting vector to obtain the result in plain form

[1] `https://search-tug.obvsg.at/primo-explore/search?vid=TUG`
[2] https://www.npmjs.com/package/psa-lib
[3] https://github.com/Safe-DEED/PSA

2.1.1 Logical separation

To understand the mechanics and, in further consequence, the proper application of the PSA library, it is most helpful to separate the library into logical components. The constituents are states, environments, inputs, and outputs. A depiction of the mentioned separation is given in Figure 1

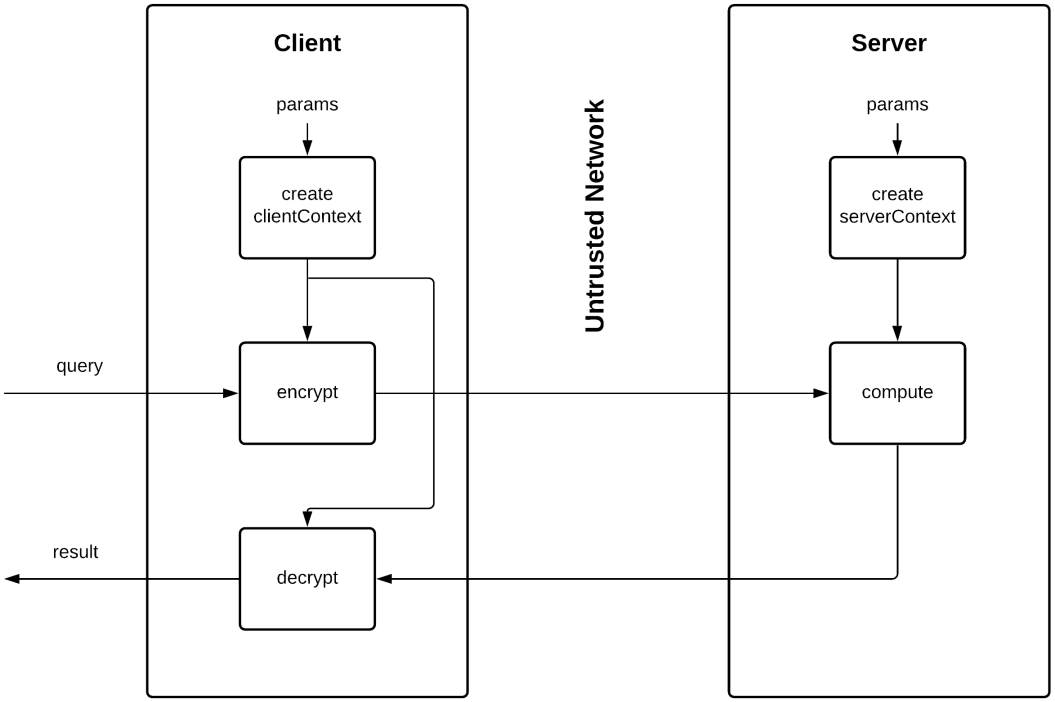


Figure 1: The PSA library seperated into its logical components.

States. There is a direct control flow from the start of the protocol to the end on a very high abstraction level. In between, there are the following states which mainly depend on a previous state:

1. Context Creation
2. Encryption
3. Computation
4. Decryption

Every state is associated with executing a set of tasks that are not crucial at this abstraction level. Although all state transitions happen sequentially, they do not occur in the same environment.

Environments. An environment is a context in which a particular instance of the library is running. There are two contexts, Client and Server, which need to be separately initialized since the objects obtained from the initialization carry different data. During initialization in the client context, a secret key pair is generated among other keys like the Galois keys and re-linearization keys. These keys are later needed in the process and are not generated on the Server's end. Running client functions with a server context object would result in an error. Due to this slightly different initialization step and inherently the functions that can be run in a specific context, we differentiate between these two contexts. The Server takes a whole different part in the protocol compared to the Client, and the underlying executing platform can be technically different. The user only needs to know whether a particular instance of the library should run in a client or server environment. The library then handles the technical discrepancy. The user of the library is not required to run the code in the browser or on a Node.js platform. For technical details, we refer to the thesis.

Inputs and Outputs. Since PSA is a protocol that demands interaction between two parties, state transitions and their outputs depend on specific inputs. Obtaining the required inputs need inter-environment communication, which is often performed over untrusted networks. The library does not handle the network part since this is highly dependent on the application and should remain the library user's responsibility. The library's function calls yield objects which should be marshaled and sent to the opposing party.

Parameters. Choosing suitable parameters for the PSA library is not trivial and requires careful consideration of the application's characteristics and knowledge of advanced cryptographic primitives. In the thesis, there is a technical section describing the process of choosing parameters for non-cryptographic experts (but still a technical section).

2.2 Privacy-Preserving Questionnaire

We have explained the use case and the motivation of the privacy-preserving questionnaire arising from the DVC (WP4) in Safe-DEED D5.8. Here, we only describe the use case with regards to the PSA library.

Subject A, who could be an institution or an individual who, wants to acquire a quality rating for a dataset. Eurecat does offer a *private rating process*, and A would like to make use of it. A visits Eurecat's website and fills out the questionnaire that holds a couple of questions about the data. After checking some boxes and assigning values, A clicks the go-button and waits for a result. Meanwhile, the questionnaire is converted into numerical form by applying a function that maps the answers to an array (vector) of fixed numbers. After that, the array is locally encrypted in A's browser and sent to the Server running at Eurecat's end. The Server holds a scoring table that maps every answer to a score. All scores are summed up and aggregated in a final score. Note that the scoring table is matched with the answers homomorphically, that is, in the encrypted domain. The Server does not learn anything about the answers. The scoring table is the matrix used to compute a linear transformation (assigning a score for each answer) on A's encrypted array. The result is still encrypted and sent back to A, where it gets decrypted by A and is ready to be further inspected.

Eurecat does not learn anything, not even the final rating, although their users are the ones who provide the rating in the first place. This procedure is depicted in Figure 2. It seems

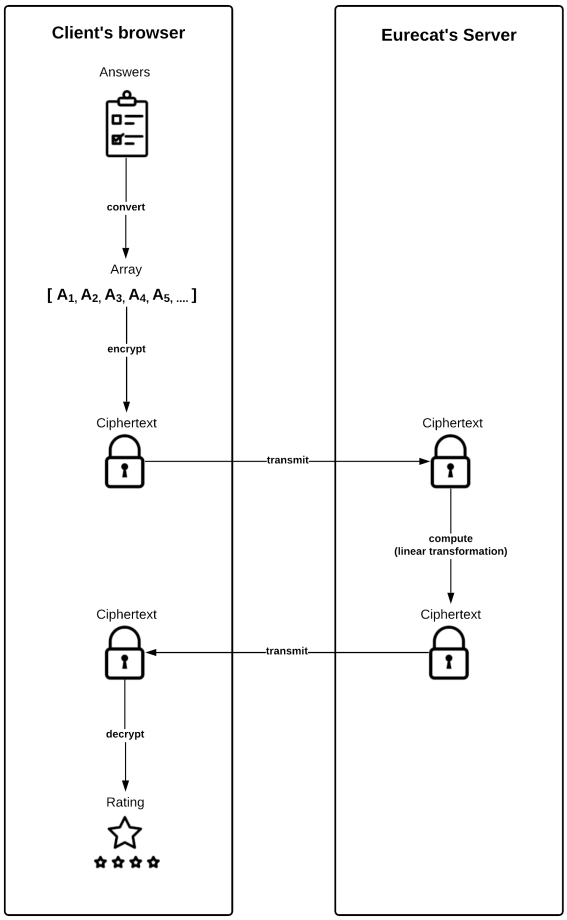


Figure 2: A high-level visualization of the private rating obtaining process at Eurecat.

controversial to rate data without even knowing the rating in the end. This is the power of a system based on Homomorphic Encryption. Eurecat has a fixed matrix that is used to apply a linear transformation on A's answer vector. Not all of A's answers have the same impact on the value of the data. Therefore some answers add more value to the overall quality. Thus, the transformation could be seen as a way of applying weights to every answer and summing them up in the end. This transformation entirely happens in the encrypted domain. Eurecat knows how the answers at A's end are formated and encrypted before transmission and relies on this form when processing the data at the server end. The most crucial part of this whole system is to define the matrix such that it will produce a reasonable rating for a set of answers.

2.3 Benchmarks

The PSA library's runtime solely depends on the number of encrypted multiplications between the input vector and the matrix. So we will focus on the vector-matrix multiplication's perfor-

mance. All other tasks, including the client-side's homomorphic encryption, are negligible in terms of runtime.

The JavaScript version is slower by a factor of six compared to the C++ version, see (Figure 3). However, data sets of similar sizes as those in Safe-DEED, this overhead does not pose a problem and, is general outweighed by the easy integration of JavaScript into existing applications. For example, around 8000 questionnaires with around 16000 questions could be processed in one matrix multiplication (10 minutes). One could also use a smaller matrix for the questionnaire to reduce the runtime, but for the benchmark, we wanted to show the limits of the PSA library. For a more detailed discussion of the benchmarks, please consult the master thesis.

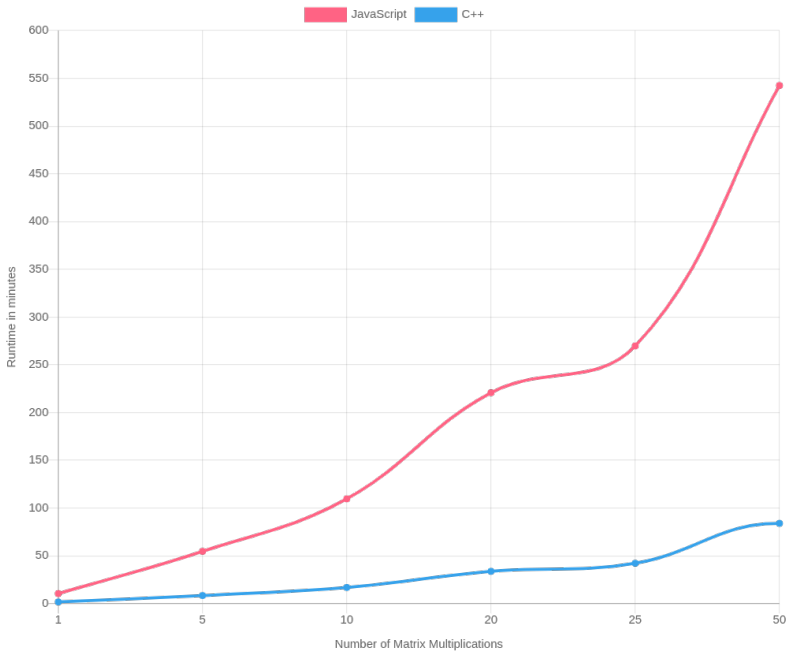


Figure 3: PSA runtime of JavaScript implementation compared to C++; matrix of size 16384×8912

2.4 Further Use Case: Risk Assessment

PSA can be used for risk assessments. It improves the companies' ability to estimate investment risks. The parties in this scenario are, on the one hand, a financial institution (server) that is aware of the credit standing and financial stability of a large pool of customers, and on the other hand, a company (client) that plans to make an investment.

In this case, the financial institution has knowledge about the credit scores of various stakeholders. The company can then ask about the credit score of stakeholders that are of interest in the light of its upcoming investment. Thereby the company can better predict the risk of the investment. This process is depicted in Figure 4. The benefit of using the PSA protocol is that the company can keep its investment idea a secret on the one side, while on the other side, that the financial institution only gives away aggregated information about stakeholders.

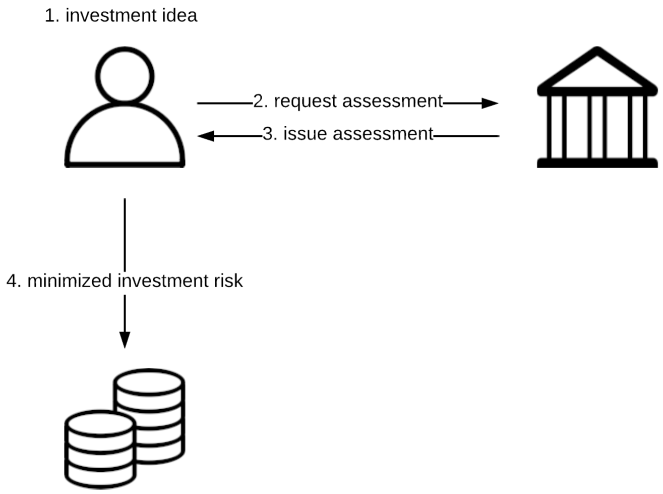


Figure 4: PSA applied to the area of investments.

2.4.1 Installing

As we exceeded the file size limits, the code is available on Zenodo at the following location `https://zenodo.org/record/4742122`.

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:

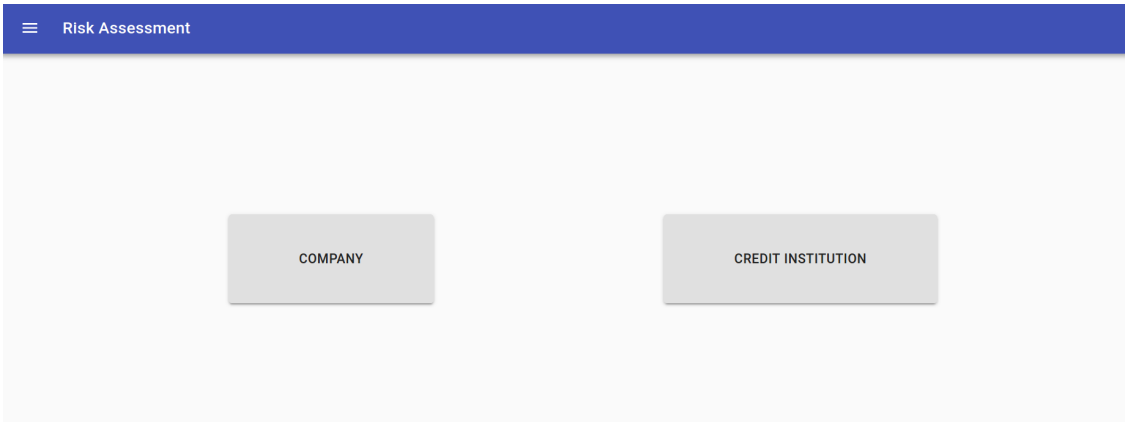
- `npm run init`
- `npm run dev`

After that, the web app is launched in the browser at `http://localhost:3000`.

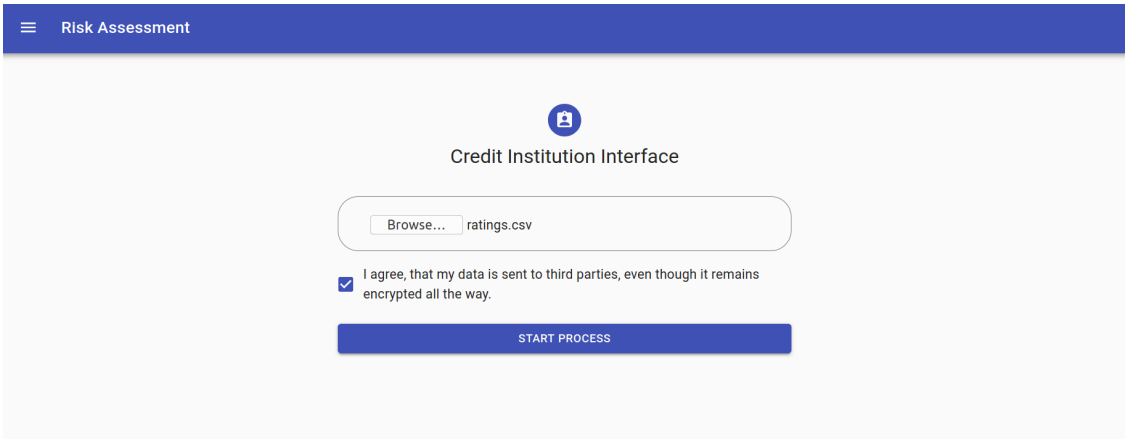
2.4.2 Running

We will now walk the reader through the process, starting 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`.

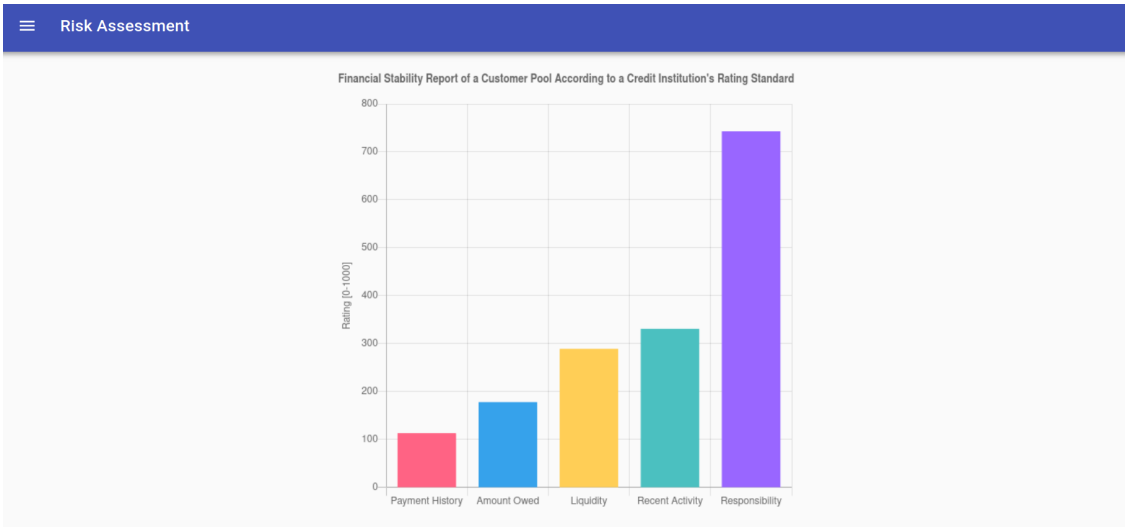
1. In the main window, we see one button for the client and one for the server. When testing the application, we recommend opening two separate browser tabs: One for the client part `http://localhost:3000/client` and one for the server part `http://localhost:3000/server`.



2. In server's tab, we browse for the credit ratings file `ratings.csv`. Then, we start the process by clicking on the button below.



3. In the client's tab we browse for the identifiers file `customers_IBANS.csv`. This file contains the identifiers for which the client is interested. After clicking start, the protocol's execution begins. The aggregated ratings are displayed at the client side after termination.



3 Secure Lead-Time Based Pricing (SLTBP)

3.1 Introduction and Update

This project focused on Secure Lead-Time based pricing (SLTBP). In our setting, we have a big enterprise vendor and many small client companies. After several iterations of improvements, the following structure emerged:

The protocol accepts an arbitrary number of input products. Each product information contains a delivery date, a price, and a volume. Both the clients and the vendor have to provide this information for each product. Then, all parties evaluate every individual product in two stages.

In the first stage, the clients aggregate their input securely. They compute the sum of their volumes on the one hand and the complete price offer for it on the other hand. The aggregation step also determines the fastest delivery time (i.e., the lowest date value of the client inputs). The protocol terminates if the volume entry of the vendor is smaller than the aggregated amount of the clients.

The second phase of the protocol determines the pricing of the product. In LTBP, the price is determined by the delivery date. The use case partner from WP7 provided four possible candidate functions to determine the pricing of a product. All of them calculate the difference in delivery time and derive a price premium from it. Finally, the parties check whether the aggregated price offer meets the requirements imposed by the delivery time and the original price.

3.2 Implementation

We elaborated on the toolchain of maven and Fresco in the previous deliverable D5.4. However, now we want to give an overview of the implementation. Focusing on the program's recent changes, we will describe the class structure, the current versions of our algorithms, and additional building blocks implemented. The current version of the program focuses on high flexibility in the price calculation. We did not have access to the final Lead-Time Based Pricing algorithm at the time of implementation. Hence, we decided to design a generic approach to make pricing algorithms interchangeable. As extensions of this generic base class, we also provide four basic pricing implementations based on a previous deliverable by WP7. Since we cannot disclose deliverable 7.5, we will cover the necessary information.

In addition to the protocol itself, we developed several helper classes for ease of implementation in MPC. Finally, we added TLS support to have realistic test results in the WAN setting.

3.2.1 Class Structure

In contrast to previous versions, the project consists of two packages: the Application and the Util package. The Application package represents a merge of the server and client functionalities. It is sufficient to differentiate between the two at runtime, as they fundamentally have to perform the same operations. On the other hand, the Util package provides handlers for secret shared values, networking, and often used helper MPC protocols.

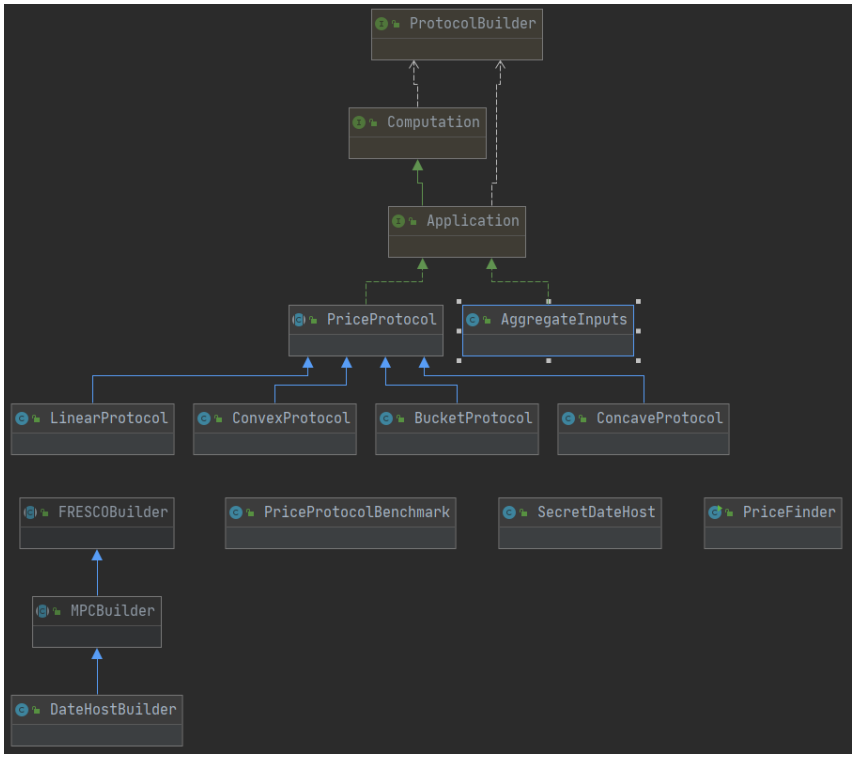


Figure 5: Class Diagram of the Application package

The Application Package. In Figure 5, we can see the classes of the application package and their inheritances. The `PriceFinder` is our entry point to this demonstrator. Here, we set up the framework and initialize the protocols, selecting the mode, in which we want to run the application. These settings include security parameters, algorithm choice, networking settings, input files, and possible command-line interface setup.

`PriceProtocol` is the previously mentioned abstract class for price calculations. One can find our implementations of the pricing protocols in the following extensions: `LinearProtocol`, `ConvexProtocol`, `BucketProtocol`, and `ConcaveProtocol`. All are named by their distinct shape of the pricing function. `AggregateInputs` contains the first phase of our protocol. There, we aggregate the orders, select the date and compare the volumes. The `Application`, `Computation`, and `ProtocolBuilder` Interfaces are part of FRESCO[4] but included for completeness. An implementation of the Application Interface represents a runnable MPC protocol.

Finally, the `DateHostBuilder` uses the builder pattern to construct a `SecretDateHost` instance. This instance then runs the aggregated input protocol and the selected price protocol. If benchmarking is set, the `PriceProtocolBenchmark` class runs all the pricing protocols and measures time and network usage.

The Utility(`util`) Package. The second package in Figure 6 contains the utility and helper classes and their inheritances.

`MultApplication` and `SIntComparator` are both small MPC helper protocols. While

[4] `https://fresco.readthedocs.io/en/latest/`

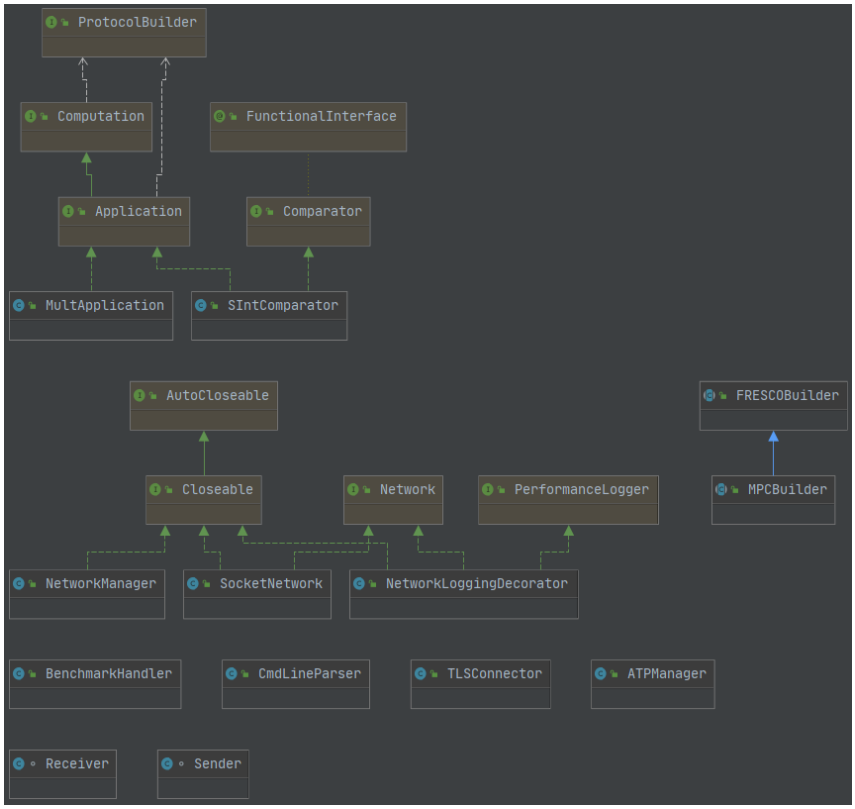


Figure 6: Class Diagram of the util package

`MultApplication` only presents a wrapper to a secret share multiplication, `SIntComparator` also implements the `Comparator` Interface. This implementation allows us to do sorting operations on data structures using the Java Comparator interface.

Next, there are the networking classes

- `NetworkManager`
- `SocketNetwork`
- `NetworkLoggingDecorator`
- `TLSConnector`
- `Sender`
- `Receiver`

The `NetworkLoggingDecorator` uses the decorator pattern and tracks data usage for benchmarking. The more practical `NetworkManager` acts as a container class for different network instances. When creating a new network, the workflow is as follows. First, the `NetworkManager` creates a `SocketNetwork` object. This object then creates the `TLSConnector` to connect to all the parties in the network configuration. The `TLSConnector` creates sockets for each peer and

also verifies them according to the pre-shared certificates. Finally, the `SocketNetwork` spawns a `Sender` and a `Receiver` thread for each other party.

The `BenchmarkHandler` and `CmdLineParser` are pure utility classes. The handler allows the generation of different timer instances and retrieves the networking data from the decorator. It generates the JSON output required by the benchmarking platform. The `CmdLineParser` is currently not used. In the earlier versions, we passed all the settings, be it security or input, as a command-line argument. As the list grew, we switched to file-based data input and settings determined at compile time. However, we kept the functionality if we want to switch back to a more versatile program. (i.e., if we're going to run the program with different security settings without wanting to recompile)

We developed the builder abstraction in the form of `FRESCOBuilder` and `MPCBuilder` for earlier versions of FRESCO. Back then, a large amount of boilerplate code was necessary to start the framework with the secure MASCOT [5] preprocessing strategy and Naor Pinkas Oblivious transfer[5] enabled. These classes are still in use in the current version, as the setup of our implementation relies on their structure. The `MPCBuilder` then initializes fields relevant to the protocol.

3.2.2 The MPC Protocol

We based our demonstrator on FRESCO. In this chapter, we want to state the concrete instantiation of the framework we use. As mentioned before, we work in the arithmetic domain. More precisely, we use the SPDZ [3] protocol suite, with MASCOT preprocessing and Naor Pinkas oblivious transfer. This setup gives us malicious security with a significant performance penalty.

FRESCO does not provide the functionality to switch between binary and arithmetic data types. Hence, we were limited to either SPDZ or SPDZ2k. And since comparisons are only possible with SPDZ, we made our choice.

The setup of our protocol happens in the `PriceFinder` class. The following list shows the essential parameters for setting up the correct instance:

[5] `http://www.pinkas.net/ot.html`

Type	Name	Description
`boolean`	`logging`	Enables debug output throughout the application
`boolean`	`debug`	Runs all pricing protocols in plain to check results
`boolean`	`benchmark`	Runs all pricing protocols after one another and tracks their time and network usage
`Enum`	`EvaluationProtocol`	Defines the pricing protocol to run, if benchmark is disabled
`Enum`	`PreprocessingStrategy`	Defines source of multiplication triples. Must be set to MASCOT for secure computations
`Enum`	`ObliviousTransferProtocol`	Must be set to Naor, to enable the secure Naor Pinkas OT protocol
`int`	`maxBitLength`	Defines the maximum bit length of secret shared values
`int`	`modBitLength`	Defines the bit length of the modulus

3.2.3 Pricing Protocols

The pricing algorithms represent the core part of our protocol. As mentioned above, we have implemented four different versions to compare them better. Our partners presented the formal specification of the algorithms in deliverable D7.5. We then jointly worked on making the pricing function MPC-friendly, i.e., reducing computational steps that are hard to compute in MPC while not using the utility of the pricing functions.

We decided to implement the unoptimized pricing function to use it as a baseline for evaluating the performance of the other functions. This way, we can make generic performance statements about the different types. These more primitive functions serve as building blocks and can be combined to fit the needs of a vendor.

The performance impact of the inputs is also worth mentioning. If the ordered volume or date is higher than the server volume or date, the protocol terminates early. However, if the whole protocol runs, the participants' input does not influence the runtime or network traffic. If the input would influence either of those metrics, an attacker could infer information about that input. Such a side-channel cannot be possible in an information theoretic setting as in SPDZ [2].

- The common parameters of each protocol are the following:
  SDT: standard delivery time. The time input of the server.
  OLT: ordered lead time. The common order time of the clients. Must be greater than 0 and smaller than **SDT**
  $\mathbf{P}_{host}$: The price input of the host, for the standard delivery time.
  $\mathbf{P}_{new}$: The new price calculated regarding the shorter delivery time.

- **Linear:** The linear protocol in D7.5 presents a formula calculating the price premium.

Here, we only made adaptations to the order of execution.

$$P_{new} = \frac{(SDT - OLT)P_{host}}{SLT} + P_{host}$$

- **Concave:** Since we perform our computations in the integer domain, we made some changes to the logarithm-based algorithms. The newly introduced exponentiation step allows for higher precision in the price calculation, as intermediate results are not rounded as much. We can still show that our adaptations are not changing the original formula from D7.5.

$$\begin{aligned} P_{new} &= \frac{\log((100(SDT - OLT))^{10}) - \log(SDT^{10})}{20} P_{host} \\ &= \frac{10\log(100(SDT - OLT)) - 10\log(SDT)}{20} P_{host} \\ &= \big(\log(100(SDT - OLT)) - \log(SDT)\big) P_{host} \cdot 0.5 \\ &= \log\left(100\frac{SDT - OLT}{SDT}\right) P_{host} \cdot 0.5 \end{aligned}$$

  We also assert a fixed ceiling of 100% price increase.

- **Convex:** Beginning at the formula we use in our implementation, we show that it is equal to the one presented in D7.5.

$$\begin{aligned} P_{new} &= \frac{\log(SDT^{10}) - \log(OLT^{10})}{20} P_{host} \\ &= \frac{10\log(SDT) - 10\log(OLT)}{20} P_{host} \\ &= \big(\log(SDT) - \log(OLT)\big) P_{host} \cdot 0.5 \\ &= \log\left(\frac{SDT}{OLT}\right) P_{host} \cdot 0.5 \end{aligned}$$

- **Bucket:** The Bucket protocol sorts the delivery speedups into distinct buckets. In the bucket protocol we left out the final entry in the table in Figure 6 of D7.5(The table is depicted below). We did this, as the delivery cannot be more than 100% faster and we excluded the case of same day delivery (i.e., **OLT** being zero).

  This leaves us with eight distinct cases. We perform a binary search with three comparisons. In the end, only the bucket is revealed to all participants.

Buckets	Price Premium
$0\% < x < 10\%$	0%
$10\% < x < 20\%$	2%
$20\% < x < 35\%$	5%
$35\% < x < 50\%$	10%
$50\% < x < 65\%$	20%
$65\% < x < 80\%$	40%
$80\% < x < 90\%$	80%
$90\% < x < 100\%$	90%
$100\% < x$	100%

3.2.4 Installing the Toolchain

This project was developed with Ubuntu. In the root folder, there is a Dockerfile, simplifying the installation process. When Docker is installed and added to your PATH, the following commands will run the program. Note: depending on the installation of docker, the following commands might have to be executed with root.

1. `docker build -t sltbp .` This will build and install the Docker image
2. `docker run -i sltbp` This will run the Docker image using the docker daemon in interactive mode.
3. `make all` This will install and run the setup specified in the Makefile, inside the Docker container.

To run the program from the command line, we suggest installing the following programs with: `sudo apt-get install -y <program>`

- `openjdk-11-jdk`
- `maven`
- `make`

After installing the requirements, go to the root directory and run the program with `make all` This will compile the program, install all dependencies, create the file structure for testing and run the setup specified in the Makefile.

3.3 Running different setups

The Makefile mentioned above helps us simulate multiple parties locally. In this chapter, we give a quick overview of the different settings.

Network Configurations The SLTBP protocol uses a TLS peer-to-peer socket network. Hence, we need to set up our keys and certificates appropriately before we can start. In the directory `SecStores`, we provide a small shell script that does this job. We need to run `bash genstores.sh <no of parties>`. The following procedure creates Java keystores and certificates for all parties and a Java truststore which all parties share. The script then copies the key materials in the resources folder of our utils package, making it accessible at runtime. The second major component of our networking configuration is the `NetworkConfig.json` found in the servers directory structure. The config file of each participant must contain an entry for each party, including itself. The `id`s are defined to be given in ascending order, while the host has `id = 1`. An entry must further contain the `IP` and `Port` information of the other parties. Finally, the field `my_id` signals that an entry describes my network properties.

Protocol input We have already touched on the topic of protocol inputs in the introduction. Now we want to give a quick overview of how we can change these settings. Next to the Network Configuration file, each party expects an ATP Unit JSON file, as seen in the server's directory structure. In general, the protocol supports an arbitrary number of input units. However, each party must have the same amount of product entries and the same sales positions. In other words, each participant must bid for each product, and the sales position is the identifier of a product. When meeting these preconditions, everyone can set their delivery Date in days, their amount, and their price as strings of integers. The date value cannot be zero, as the design excluded same-day delivery.

3.4 Benchmarking

Next to local machine tests, we wanted to analyze our protocol in the LAN and WAN settings. The LAN setting helps us to test the performance in general, while the WAN setting really simulates a real-world scenario. We were particularly interested in the runtime and network traffic impacts of the number of parties and the number of products for both configurations. On the one hand, we Further, we want to compare the different shapes of pricing functions considering these metrics.

Results: Running our tests showed precise results. When we increase the number of players, this results in a quadratic increase in runtime and network usage. In comparison, the runtime and network impact of the products in the protocol is linear. In the following table, we give an overview of our results. We show the resulting graphs further down (fig. 7 to fig. 14).

Networking	No. of Players	No. of Products	max Runtime	max traffic
LAN	3…10	1	10h	14GB
LAN	3	1…10	2h	16GB
WAN	3…6	1	6h	6GB
WAN	3	1…10	12h	16GB

Interpreting the table In the table, we see the three different ways in which we performed our benchmarks. Firstly, we ran all tests both in the LAN and in the WAN setting. Secondly, when measuring the impact of the players, we increased the number of players and kept the

number of products constant. We started at the internal fixed minimum of three players since we defined this threshold for security reasons. In a two-party setting, the vendor party could infer the input of the vendee after the result is disclosed. In the LAN setting, we increased the number of players up to ten to gain a good impression. However, we ran into an internal timeout in one of our sub-protocols at seven players with the WAN configuration. But, we can already see the expected impact on runtime and network traffic. Lastly, we tested the products' input by keeping the player number at three and increasing the products from one to ten. We computed all of our experiments on a Xeon(R) CPU E5-2660 v3 @ 2.60GHz. Each player of the protocol had a core reserved. We created an artificial WAN setup on the server.

Further improvements In our implementation, we allow for a wide range of players and a wide range in the number of products. It is possible to increase performance by fine-tuning the protocols to a specific context. Nevertheless, the quadratic growth property will remain.

The impact of the individual pricing protocols is smaller than expected. However, the differences between the protocols are less impactful since they share the same aggregation and evaluation step. As we can see in our graphs, the Concave and the Covex protocol are nearly identical in terms of runtime and network traffic. A little bit below is the bucket protocol and by far the fastest is the linear pricing calculation.

In conclusion, we see several ways to improve the performance of this proof-of-concept demonstrator. FRESCO allowed us to develop a functioning demonstrator quickly. However, we were very limited in the algorithms we could choose. For our use case, we could only use SPDZ, with MASCOT preprocessing and NAOR Pinkas OT. Secondly, Java as a platform presents an overhead as well. Working with a more low-level language like C++ would be beneficial ([4]). Finally, as mentioned above, after designing a final protocol for price calculation, the code can be fine-tuned to meet specific requirements.

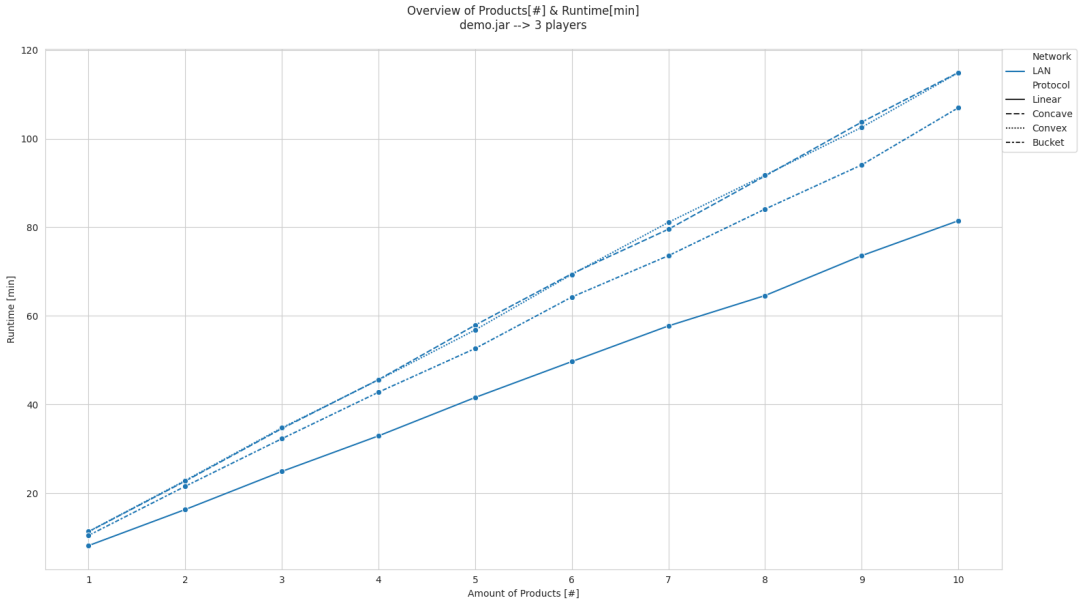


Figure 7: Impact of products on runtime (LAN)

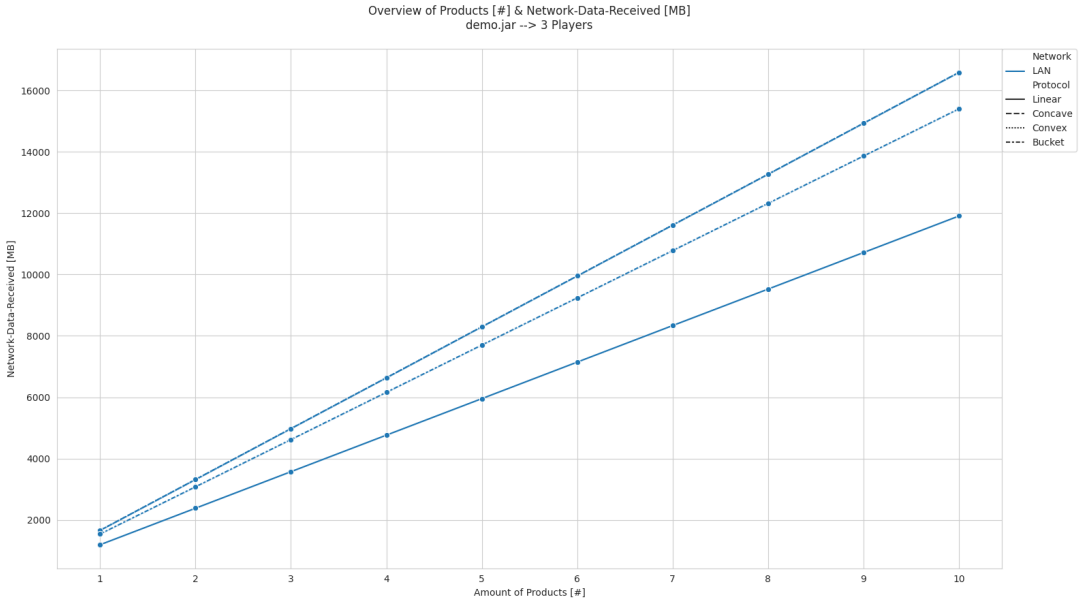


Figure 8: Impact of products on networking (LAN)

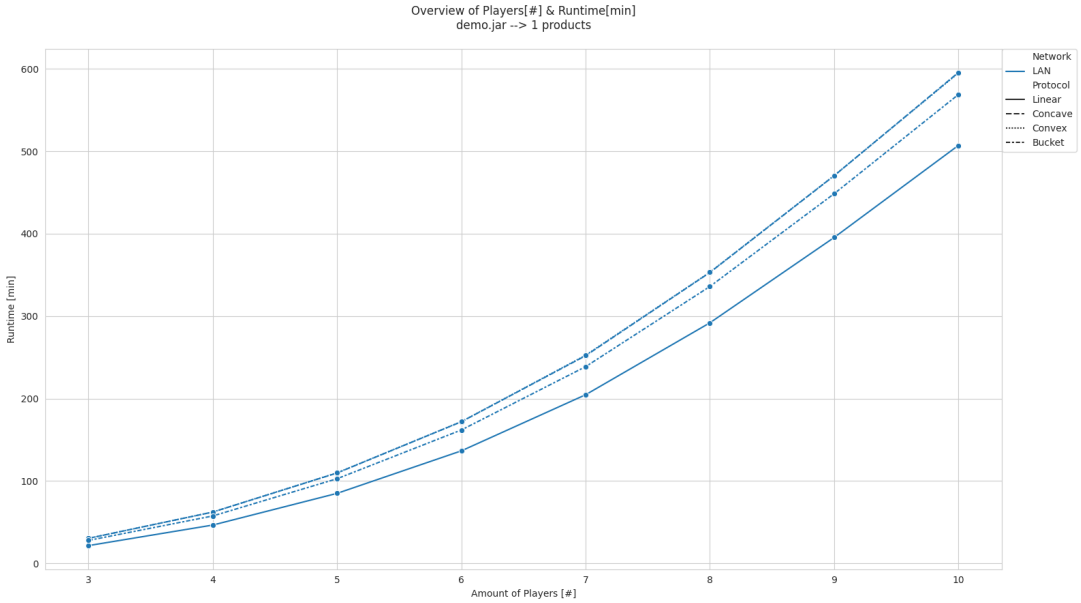


Figure 9: Impact of players on runtime (LAN)

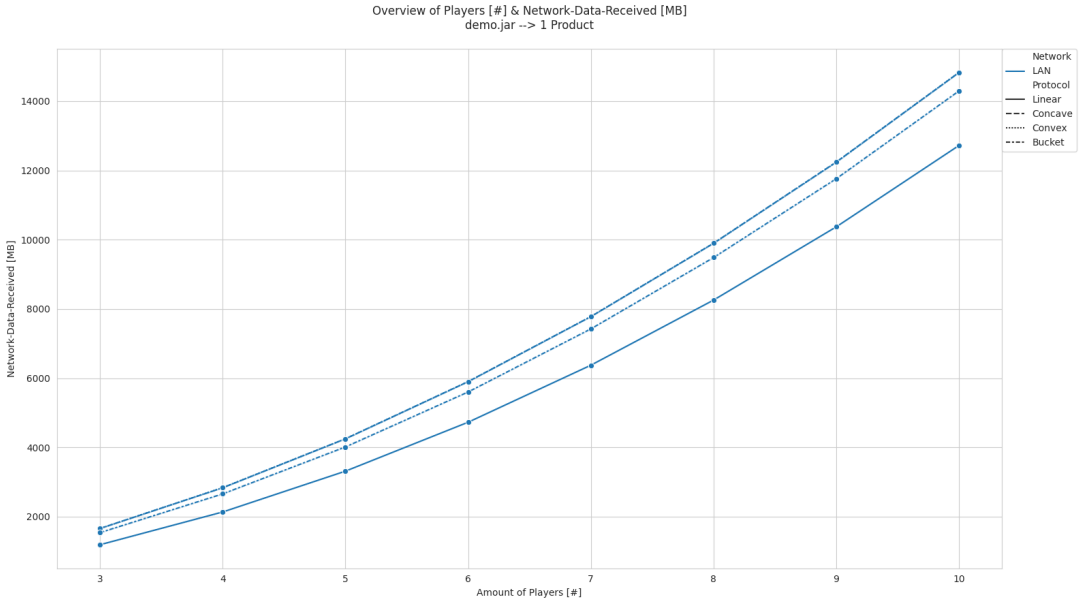


Figure 10: Impact of players on networking (LAN)

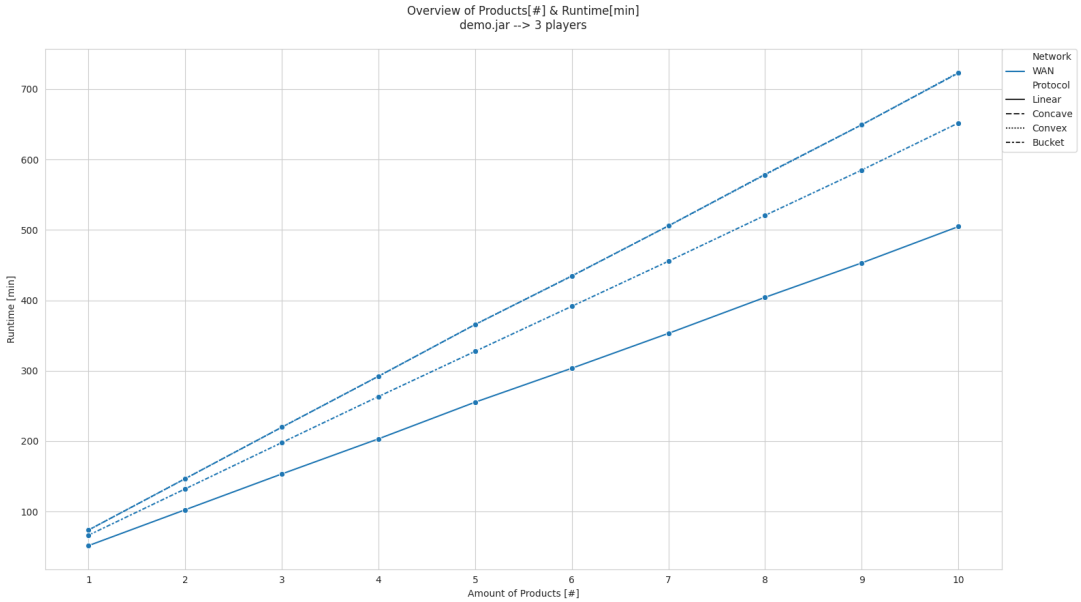


Figure 11: Impact of products on runtime (WAN)

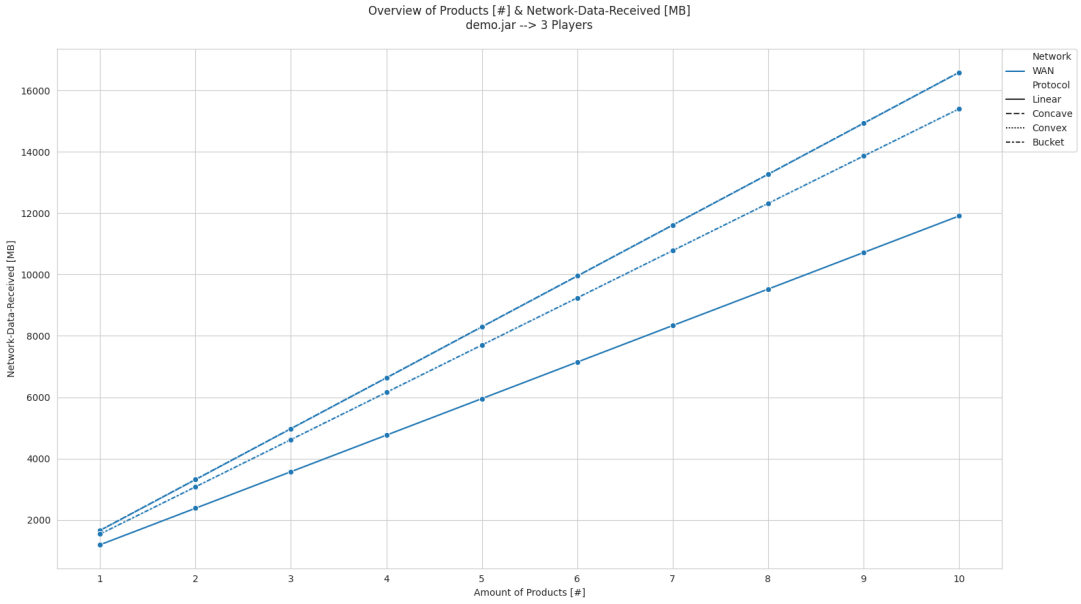


Figure 12: Impact of products on networking (WAN)

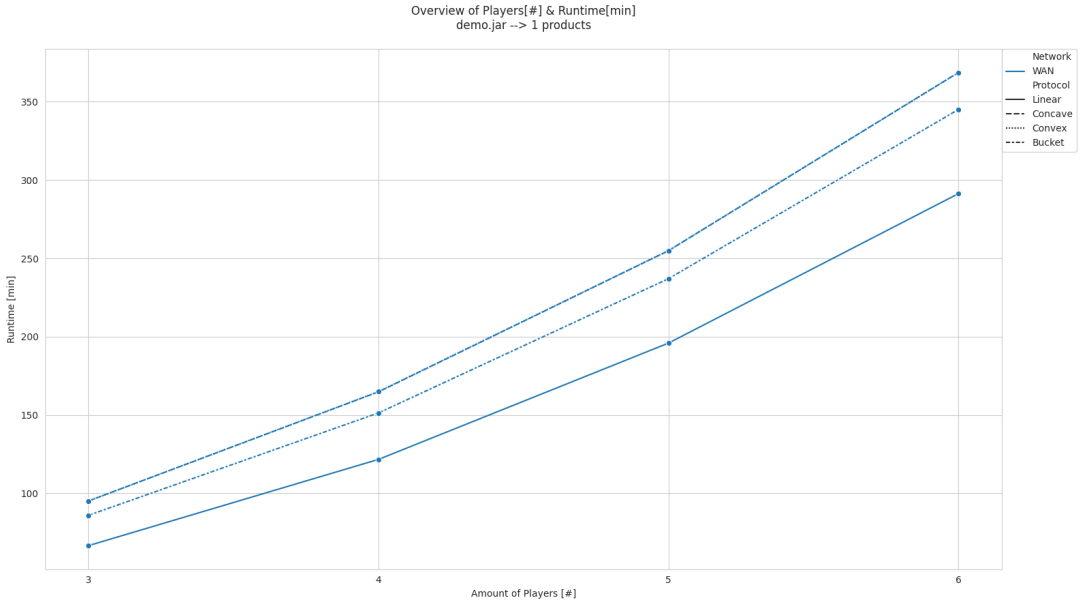


Figure 13: Impact of players on runtime (WAN)

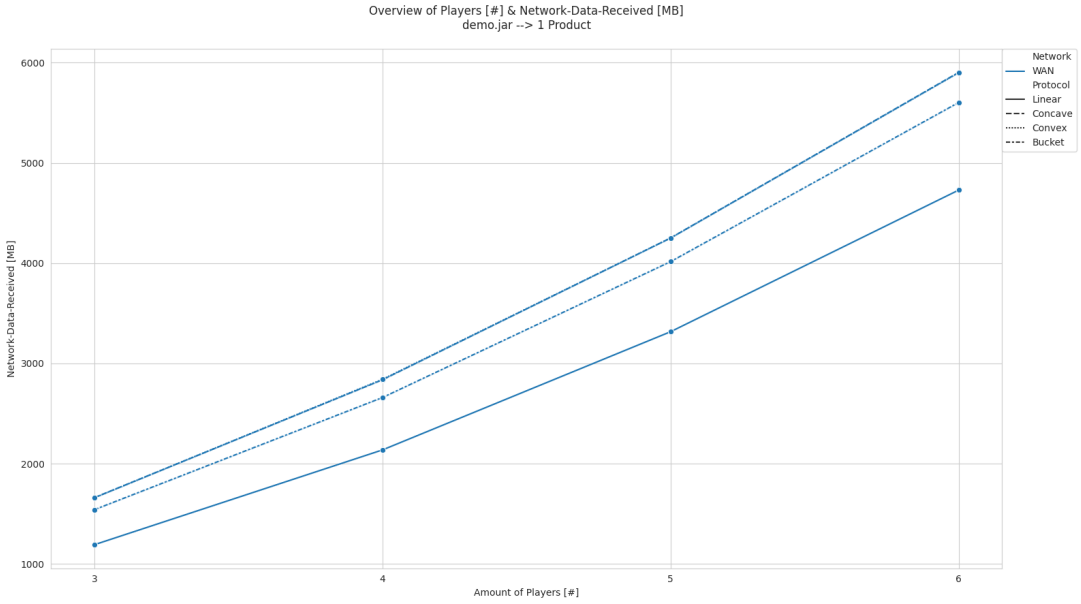


Figure 14: Impact of players on Networking (WAN)

4 Conclusion

This deliverable contains two demonstrations of secure computation protocols. In this accompanying report, we described their functionalities and their performance, respectively, their scalability. Concerning their applicability to real-world scenarios, our conclusion is: it depends.

We can see that the light-weight PSA library can even be used on the web to perform secure computations on the client-side. As we move towards more complex and use-case-specific computations, the overhead of MPC is clearly felt. In such use cases, we recommend checking if

semi-honest trust assumptions are sufficient. This could be the case if the participants have long-term interests or there exists complementary legal contracts. Nevertheless, our demonstrator shows that secure computations can be efficient enough. Although complex scenarios still need careful design and engineering.

For the remaining time of the project, we will focus on assisting the use case partners in integrating the two secure computation protocols into their systems.

5 References

[1] Grass Alexander. Evaluation of private selective aggregation in the web and its application in real-world scenarios, 2021.

[2] I. Damgard, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. `https://eprint.iacr.org/2011/535`.

[3] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[4] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. `https://eprint.iacr.org/2020/521`.

[5] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.