German Medical Data Sciences 2024
R. Röhrig et al. (Eds.)

doi:10.3233/SHTI240863

Simplifying Multiparty Computation: A Client-Driven Metaprotocol for Federated Secure Computing

Johanna Schwinn[a,1], Hendrik Ballhausen[b], Seyedmostafa Sheikhalishahi [a], Matthaeus Morhart [a], Mathias Kaspar[a], and Ludwig Christian Hinske [a]
[a] *Digital Medicine, University Hospital of Augsburg, Augsburg, Germany*
[b] *Department of Radiation Oncology, LMU University Hospital, LMU Munich*
ORCiD ID: Johanna Schwinn, https://orcid.org/0009-0000-3107-1619

Introduction. Secure Multi-Party Computation (SMPC) offers a powerful tool for collaborative healthcare research while preserving patient data privacy. **State of the art.** However, existing SMPC frameworks often require separate executions for each desired computation and measurement period, limiting user flexibility. **Concept.** This research explores the potential of a client-driven metaprotocol for the Federated Secure Computing (FSC) framework and its SImple Multiparty ComputatiON (SIMON) protocol as a step towards more flexible SMPC solutions. **Implementation.** This client-driven metaprotocol empowers users to specify and execute multiple calculations across diverse measurement periods within a single client-side code execution. This eliminates the need for repeated code executions and streamlines the analysis process. The metaprotocol offers a user-friendly interface, enabling researchers with limited cryptography expertise to leverage the power of SMPC for complex healthcare analyses. **Lessons learned.** We evaluate the performance of the client-driven metaprotocol against a baseline iterative approach. Our evaluation demonstrates performance improvements compared to traditional iterative approaches, making this metaprotocol a valuable tool for advancing secure and efficient collaborative healthcare research.

Keywords. Medical Research, Cryptography, Health Information Exchange, Confidentiality

1. Introduction

Extracting valuable insights from large and diverse healthcare datasets is crucial for advancing medical research and improving patient care. However, strict data privacy regulations such as the General Data Protection Regulation (GDPR) often restrict data sharing, hindering comprehensive analysis. Besides Federated Machine Learning, Secure Multi-Party Computation (SMPC) [1,2] emerges as a promising solution, enabling collaborative analysis of sensitive healthcare data while preserving data privacy at its source.

SMPC enables multiple institutions to collaboratively analyze sensitive healthcare data while keeping the data decentralized at each location. This approach facilitates the

[1] Johanna Schwinn, Digital Medicine, University Hospital Augsburg, Stenglinstraße 2, Augsburg, Germany; E-mail: Johanna.schwinn@uk-augsburg.de

extraction of valuable insights from large datasets without compromising patient privacy, improving healthcare research and collaboration. However, SMPC often involves rather complex tech stacks and difficult implementations. Also, the universality of most SMPC solutions incurs a heavy penalty in the runtime complexity and networking overhead.

In this work, we address some of these difficulties by proposing a novel client-driven metaprotocol for Federated Secure Computing [3] (FSC; see section 2.2 for details), a simple and lightweight SMPC system optimal for medical use cases. This metaprotocol aims to provide a more flexible and generalizable interface to FSC in healthcare research in cases with the following requirements:

- Execute multiple atomic operations within a single run.
- Analyze data across different measurement periods: The metaprotocol will accommodate data from various timeframes, such as April, March 2024, etc.
- Utilize various calculation methods: The system will be adaptable to different analytical approaches, such as calculating medians, sums, and other metrics.

2. Background

2.1. Related Work

There are different frameworks offering solutions to facilitate privacy-preserving computation between organizations (for non-cryptographic experts). DataSHIELD [4] is a framework developed to jointly compute summary statistics. The computation relies on exchanging only summary parameters between participating parties while the individual-level data remains private at its origin site.

Sharemind MPC [5] is a commercial industry-grade solution for Secure Multiparty Computation. Input data is secretly shared across compute nodes. Arbitrary computations may then be performed in a peer-to-peer network through bytecode compiled from the domain specific SecreC language [6]. Pre-built statistical analysis is also available from Rmind, an R-style scripting language.

The MP-SPDZ [7] framework is a fork of the SPDZ-2 implementation of the SPDZ MPC protocol containing a wide range of different security models and multi-party protocols, which are accessible through a Python interface. The focus is on providing a research platform to benchmark different MPC protocols. Particularly, MP-SPDZ has been applied in ML for secure inference and training of complex (neural network) models.

While MP-SPDZ and Sharemind MPC aim to provide a comprehensive solution, the goal of EasySMPC [8] is to make SMPC protocols easily accessible for non-cryptographers. EasySMPC is a privacy preserving SMPC framework focusing on usability and ease of access for non-technical personal. The framework offers desktop application with a graphical user interface to set up the computation. The communication between participating parties is executed automatically via e-mail. The underlying SMPC method for this software is Arithmetic Secret Sharing [1]. Currently, EasySMPC offers the secure computation of a sum as its only available calculation method.

2.2. Federated Secure Computing

Federated Secure Computing (FSC) [3] is an initiative and middleware that aims to provide access to Privacy-Preserving Computation (PPC) for a wider range of users. The key innovation of FSC lies in its architectural design. By separating the complexities of cryptographic protocols from the user's business logic, FSC makes it easier for users to leverage secure computing techniques without requiring in-depth knowledge of cryptography.

FSC incorporates a propaedeutic Simple Multiparty Computation protocol (SIMON) that provides various commonly used SMPC algorithms as microservices within the FSC framework. These algorithms include secure sum, secure median, matrix multiplication univariate and bivariate statistics, and more.
FSC aims to offer a standardized interface for federated analyses.

The core architectural principle of FSC is the separation of tasks between clients and servers. FSC offers flexibility in its client-server topologies, supporting configurations ranging from symmetric peer-to-peer networks to centralized client-server structures. For the propaedeutic framework SIMON, each client has its own server. Within the client-server architecture, servers function as compute nodes, executing the designated protocols. Conversely, clients act as both data and researcher nodes, supplying input data, dictating control flow, and receiving the processed results.

The server-side architecture of FSC handles secure computation, data processing, and secure communication with clients. The main mechanisms are registry, discovery, and the bus. Servers host microservices, that provide microprotocols like secure sum, matrix multiplication, and statistical calculations. The server-side implementation is built in Python.

The client-side architecture prioritizes lean design and interacts with server-side components via an API. The lightweight client-side software is available in different programming languages. Figure 1 shows the main snippet of the client-side code including a for loop to run multiple calculations within a single execution unit. A more detailed description can be found in [3]. The code is available online[2].

```
import federatedsecure.client
import shared
import time

NETWORK = {**shared.NETWORK, 'myself' : 1}

if __name__ == "__main__":
   for k, v in input_data.items():
        api = federatedsecure.client.Api(shared.URL_CLIENT1)
        microservice = api.create(protocol="Simon")
        result = microservice.compute(microprotocol= k.split('_')[-1],
                                      data=v,
                                      network=NETWORK
                                      )
        print(api.download(result))
        time.sleep(0.12)
```

Figure 1. The client takes as input a single number and specifically calls the microprotocol for the computation of a secure sum.

[2]Source Code of FSC: https://github.com/federatedsecure (accessed: 2024-04-30)

Summing up, FSC is a framework that is particularly suited for propaedeutic application and medical research. Here, universality is sacrificed for simpler development and faster execution. The architecture provides all secure computing functionality as lean premade microservices through an OpenAPI interface. A disadvantage is that only a set of fixed, pre-built microservices are available to the user.

3. Metaprotocol

Current implementations of FSC/SIMON client software often rely on calling the API multiple times for smaller computation blocks. This requires multiple clients to identify subtasks with multiple UUIDs, requiring additional client-client communication.

To address this limitation and enable efficient execution of sequences of computations involving multiple variables and metrics, we propose a novel client-driven metaprotocol. This metaprotocol acts as a higher-level abstraction, encapsulating a variable number of microprotocols within a single execution unit of client-side code. The entire sequence of computations is encapsulated into a single operation. In Python it uses an array of dictionaries including the concept name and measurement start and end time (e.g. total number of admissions in a hospital for a certain timeframe define by a start and end time), which is used to load data entry with the same name from a database (Table 1). The method item is used to define the atomic operation provided by FSC. The utilized input format is illustrated in Figure 2.

Table 1. This table demonstrates a sample format. "Exchange_date" indicates the date of the computation, "Measurement_date" reflects the measurement time. "Concept_id" references the measured concept, "Value_numeric" holds the value, and "Target_id" identifies input (0) or result (1).

id	Exchange_date	Measurement_date	Concept_id	Value_numeric	Target_id
001		2024-03-12T13:37:59	42	37	0
003	2024-06-21T13:49:59	2024-01-20T09:49:52	42	120	1

```
SHARED_INPUT= [{'concept_name': 'admissions', 'measurement_start': '2024-03-01T00:00:00',
                'measurement_stop':'2024-03-31T23:59:59' , 'method': 'SecureSum'},
               …]
```

Figure 2. This is the shared input format, which is used by all participating clients to use their data for joint calculations. "measurement_start" and "measurement_stop" define a time window for retrieving data entries. Only entries where the "measurement_date" falls within this window are retrieved.

The modified client-side code is shown in Figure 3. In addition, instead of directly calling individual microprotocols (e.g., “SecureSum”) in a sequential manner on the client side, the metaprotocol passes the number and methods of calculations on to the server to handle it.

```
import federatedsecure.client
import shared

NETWORK = {**shared.NETWORK, 'myself' : 1}

if __name__ == "__main__":
    api = federatedsecure.client.Api(shared.URL_CLIENT1)
    microservice = api.create(protocol="Simon")
    result = microservice.compute(microprotocol="Metaprotocol",
                                  data=input_data,
                                  network=NETWORK,
                                  metrics = list(input_data.keys())
                                  )
    print(api.download(result))
```

Figure 3. The client uses a new metaprotocol which calls the microprotocols defined in the shared input and takes the number of metrics to be calculated as an input argument. This code describes the extension of the code described for atomic operations in Figure 1.

3.1. Performance Evaluation

To evaluate the performance improvement of the proposed metaprotocol, we compared its execution time against a baseline iterative approach that mimics the standard method of executing individual microprotocols sequentially for each variable. The experimental setup involved four participants, each consisting of a client, server, and database encapsulated in individual Docker containers. The evaluation involved 200 runs, each consisting of a mixed workload with 10 computations: 5 executions of the Secure Sum microprotocol and 5 executions of the Statistics Frequency microprotocol (calculating the mode, i.e. the most frequent value, and the histogram of the distribution of the input values). During implementation and evaluation, we identified an issue with the iterative baseline where clients were no longer synchronized on the current task after a certain amount of time: The computation process typically involves a designated "distributor client" responsible for initiating a round of calculations and broadcasting a task definition including a unique task identifier (ID). Subsequent clients passively wait for any task ID broadcast. However, if a previous task ID exists from a prior round of calculations, waiting clients might only receive this outdated task ID and proceed with the associated computation. Once the new, relevant task ID is broadcasted, no clients remain in a waiting state as they have already processed the outdated ID. This results in a mismatch of task IDs, leaving the subsequent computations incomplete as the clients are out of sync. To avoid this discrepancy between the task definitions, a pause of 0.12 seconds had to be introduced after each calculation. Additionally, a 2-second sleep time was implemented between each run for both approaches.

4. Results

Table 2 shows the results of the performance evaluation of the metaprotocol. The overall average execution per run time for all clients combined is 6.894s with the new client-driven metaprotocol compared to 7.552s in the baseline iterative approach. With the new client-driven metaprotocol the average execution time ranged between 6.395s ± 0.396s and 7.393s ± 0.155s. Using an iterative approach without the metaprotocol, the average execution time is between 6.806s ± 0.114 and 7.801s ± 0.12.

Table 2. The results of the performance evaluation are reported per client as the mean time in seconds over 200 runs and the standard deviation. The row Iterative approach (ideal) refers to the iterative approach when accounted for the mandatory sleep time in between iterations.

	Client 1	Client 2	Client 3	Client 4
Metaprotocol	**7.393** ± 0.155	**7.391** ± 0.396	**6.395** ± 0.397	**6.395** ± 0.396
Iterative approach	7.800 ± 0.141	7.801 ± 0.12	7.800 ± 0.139	6.806 ± 0.114
Iterative approach (ideal)	7.680 ± 0.142	7.682 ± 0.12	7.681 ±0.139	6.687 ± 0.115

5. Discussion

In this work, we were able to show the viability of a simplified metaprotocol enabling the calculation of multiple different operations (e.g. a secure median) for different time periods on a network of multiple participants. This protocol is easier to use on the client side compared to the original implementation, and slightly faster compared to an iterative approach.

This work was necessary because FSC/SIMON generally reduces more complex calculations to simpler ones. For example, univariate or bivariate statistics are composed of several secure sum computations. And the secure median implementation even creates 'pipelines' dynamically. The SIMON implementation specifically provides language construct to start pipelines iteratively and collect their intermediate results in accumulators.

However, this functionality is strictly server-side. If the user needed to do a complex calculation that was composed of many small microservices, they would have to do so in client code. This has two shortcomings: First, the complexity is kept client-side rather than offloaded to the server, which is against the FSC design philosophy. Second, many small calls to the API cause an excessive networking overhead.

In this solution, we have shown how to implement a metaprotocol that is client-driven (i.e. the user can upload their complex data in one go) and then distribute the computational tasks to the server side. In this way, the client code is kept clean and simple to the user, e.g., a data scientist.

The results of the performance evaluation show an advantage over the iterative approach even when the required sleep time has been accounted for. A more important advantage of the metaprotocol is its ability to prevent task desynchronization among participating clients.

Next steps in the further development of a more flexible SMPC service involve a dynamic exception handling that allows clients to suspend their participation for several rounds if they are unable to provide the agreed-on input and resume their participation if they have the required input data for a computation.

6. Conclusion

We developed and evaluated a client-driven metaprotocol for SMPC that outperforms the traditional iterative approach. Our metaprotocol:

• Offers a solution to execute multiple atomic operations with different mathematical operations over various measurement periods
• Achieves an overall average execution time of 6.894s, compared to 7.552s for the iterative approach
• Prevents task desynchronization among participating clients
• Offloads computational complexity to the server-side, simplifying client code and reducing networking overhead.

Declarations

Conflict of Interest: The authors declare that there is no conflict of interest.

Contributions of the authors: Conceptualization and Investigation: CH, HB, MK, JS; Methodology: JS, MK; Software: JS; Validation: JS; Writing - Original Draft: JS; Writing - Review & Editing: All; Supervision: CH, MK; Funding acquisition: CH, HB.

Acknowledgement: This study was funded by the German Ministry of Education and Research (BMBF) (grant number #01ZZ2005) and Stifterverband (grant number H110 5114 5132 36534). The funder played no role in study design, data collection, analysis and interpretation of data, the writing of this manuscript, or the decision to publish.

References

[1] Shamir A. How to share a secret. Commun ACM 1979;22:612–3. doi:10.1145/359168.359176.
[2] Yao AC-C. How to generate and exchange secrets. 27th Annu. Symp. Found. Comput. Sci. Sfcs 1986, 1986, p. 162–7. doi:10.1109/SFCS.1986.25.
[3] Ballhausen H, Hinske LC. Federated Secure Computing. Informatics 2023;10:83. doi:10.3390/informatics10040083.
[4] Wolfson M, Wallace SE, Masca N, Rowe G, Sheehan NA, Ferretti V, et al. DataSHIELD: resolving a conflict in contemporary bioscience—performing a pooled analysis of individual-level data without sharing the data. Int J Epidemiol 2010;39:1372–82. doi:10.1093/ije/dyq111.
[5] Bogdanov D, Laur S, Willemson J. Sharemind: A Framework for Fast Privacy-Preserving Computations. In: Jajodia S, Lopez J, editors. Comput. Secur. - ESORICS 2008, Berlin, Heidelberg: Springer; 2008, p. 192–206. doi:10.1007/978-3-540-88313-5_13.
[6] Jagomägis R. SecreC: a privacy-aware programming language with applications in data mining. University of Tartu, 2010.
[7] Keller M. MP-SPDZ: A Versatile Framework for Multi-Party Computation. Proc. 2020 ACM SIGSAC Conf. Comput. Commun. Secur., New York, NY, USA: Association for Computing Machinery; 2020, p. 1575–90. doi:10.1145/3372297.3417872.
[8] Wirth FN, Kussel T, Müller A, Hamacher K, Prasser F. EasySMPC: a simple but powerful no-code tool for practical secure multiparty computation. BMC Bioinformatics 2022;23:531. doi:10.1186/s12859-022-05044-8.