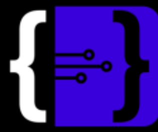


Volume 3, No. 1, January/December, 2022

Secure Federated Computing

Thomas Mason[1]

[1] University of Queensland - Australia,thomasmason40@outlook.es, https://orcid.org/0009-0001-2108-8406

ABSTRACT

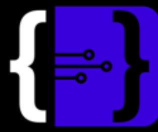
The concept of safeguarding privacy in computations, known as Privacy-Preserving Computation (PPC), involves the encrypted processing of confidential data. Despite its theoretical advantages, the intricate technology poses significant practical entry barriers. In this context, we establish design objectives and principles for a middleware that centralizes demanding cryptographic processes on the server side while offering a user-friendly interface for client-side application developers. The resultant framework, termed "Federated Secure Computing," delegates computationally intensive tasks to the server, segregating cryptographic concerns from business logic. Through an Open API 3.0 definition, it delivers microservices and accommodates various protocols via self-discovered plugins. Notably, it demands only minimal DevSecOps capabilities, ensuring simplicity and security. Moreover, its compact size makes it suitable for deployment in the Internet of Things (IoT) and introductory scenarios on consumer hardware. Benchmarks for calculations using a Secure Multiparty Computation (SMPC) protocol, applied to both vertically and horizontally partitioned data, reveal swift runtimes in seconds across dedicated workstations and IoT devices like Raspberry Pi or smartphones. The reference implementation is accessible as free and open-source software under the MIT license.

Keywords: *confidentiality-preserving computation; edge computing; collaborative computing; cryptology; secure collaborative computation*

Copyright: © 2022 by the authors. This article is an open access article distributed under the terms and conditions of the Creative Commons

Received:
07 April, 2022

Accepted for publication:
20 May, 2022


INFOTECH
Journal of Applied Sciences, Informatics, Computing and Knowledge Society
Volume 3, No. 1, January/December, 2022

Computación federada segura

RESUMEN

El concepto de preservación de la privacidad en las computaciones, conocido como Computación Preservadora de la Privacidad (CPP), implica el procesamiento encriptado de datos confidenciales. A pesar de sus ventajas teóricas, la tecnología intrincada presenta significativas barreras prácticas de entrada. En este contexto, establecemos objetivos y principios de diseño para un middleware que centraliza procesos criptográficos exigentes en el lado del servidor al tiempo que ofrece una interfaz fácil de usar para desarrolladores de aplicaciones en el lado del cliente. El marco resultante, denominado "Computación Segura Federada", delega tareas intensivas en cómputo al servidor, segregando las preocupaciones criptográficas de la lógica empresarial. A través de una definición de API abierta 3.0, proporciona microservicios y acomoda varios protocolos mediante complementos autodescubiertos. Notablemente, requiere solo capacidades mínimas de DevSecOps, garantizando simplicidad y seguridad. Además, su tamaño compacto lo hace adecuado para implementarse en el Internet de las cosas (IoT) y en escenarios introductorios en hardware de consumo. Las pruebas de rendimiento para cálculos utilizando un protocolo de Computación Segura Multiparte (SMPC), aplicado tanto a datos vertical como horizontalmente particionados, revelan tiempos de ejecución rápidos en segundos en estaciones de trabajo dedicadas y dispositivos IoT como Raspberry Pi o teléfonos inteligentes. La implementación de referencia está disponible como software libre y de código abierto bajo la licencia MIT.

Palabras clave: *computación preservadora de la confidencialidad; computación en el borde; computación colaborativa; criptología; computación colaborativa segura*

INTRODUCTION

Data is frequently likened to the "new oil" that propels the digital economy into the future. Both society and science rely on data to make well-informed decisions, and the reliability of such information is enhanced when drawing from diverse and independent data sources. The recognition of the value of data assets is growing among enterprises and consumers, particularly when these data are aggregated from previously inaccessible silos into extensive data lakes. However, there is a significant hurdle in openly sharing data. Companies fear divulging trade secrets to competitors, research and development agencies guard their work closely, and consumers are apprehensive about potential privacy violations and the misuse of their data to their detriment.

The pervasive concerns regarding data protection and security have transformed informational self-determination into a de facto basic human right. Consequently, although 84% of companies believe that analytics will enhance their competitive position to some extent [1], and 75% express a willingness to share their data [2], only 39% of European companies claim to engage in data sharing with others [3]. Similarly, 92% of internet users are worried about privacy [4]. In essence, there has traditionally been a tradeoff between the perceived value of data sharing and the imperative need for privacy and data protection.

Public domain data and open data offer significant societal and public economic benefits but necessitate participants relinquishing their rights to their data, with limited post-consent control. A middle ground is found in data sharing and collaboration, often governed by licenses and contracts. However, due to the transient nature of data, controlling and enforcing such arrangements can be challenging. Alternatively, data may be privately owned, with access restricted and limited by other parties. Over time, various attempts have been made to establish frameworks for data sharing that maximize benefits while minimizing drawbacks.

In the realm of "trusted third parties," a classic analog example involves a business consultant who confidentially learns the trade secrets of multiple companies and redistributes sanitized benchmarks and best practices among them. In the information age, "federated computing" seeks to integrate data from diverse sources, with results computed by involved parties in a distributed manner. Privacy-

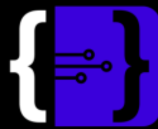
preserving computation (PPC), notably through techniques like secure multiparty computation (SMPC) [5,6], encrypts input data locally as cryptographic shares. It then performs joint computations on these shares in a peer-to-peer network to derive results known to all involved parties. SMPC is often viewed as a gold standard, offering potentially mathematically proven security even in anonymous, trustless settings with malicious parties seeking to deviate from the protocol to reveal secret inputs from other parties.

Fully Homomorphic Encryption (FHE) [7,8] encrypts input data locally before transmitting it to a public cloud, which then conducts computations solely on the encrypted data. Despite its simplicity and potency, only a limited number of algorithms are effective when applied to encrypted data. Differential Privacy (DP) [9,10] aims to control access to certain databases by managing a "privacy budget," restricting researchers from obtaining detailed information about individual data points in the database. This approach proves beneficial for scientific computations, such as those performed on medical databases to extract aggregate statistics.

For an overview of related technologies, refer to [11]. These technologies share a common challenge to varying degrees. While they excel at reconciling data sharing with privacy, they do so at the expense of increased computational overhead and operational complexity. For instance, in the case of Secure Multiparty Computation (SMPC), most frameworks attempt to offer a comprehensive yet universal solution, replacing basic arithmetic operations with exponentially growing garbled or arithmetic circuits [12,13], or costly asymmetric encryption [14–16].

Only a handful of commercially available turn-key solutions exist, such as Sharemind [17]. On the other hand, open-source frameworks, though available, are seldom industry-grade. They often depend on complex dependencies and are built on exotic tech stacks that prove cumbersome for development, operation, and security. Few are user-friendly and explicitly designed for use in introductory settings, such as EasySMPC [18].

Currently, Privacy-Preserving Computation (PPC) faces various barriers to entry. Some are on the business side, such as a lack of financial incentives and unproven business models, or are purely historical, like the absence of visible role models and successful showcases. However, numerous


INFOTECH
Journal of Applied Sciences, Informatics, Computing and Knowledge Society
Volume 3, No. 1, January/December, 2022

barriers are associated with technical challenges that make the development, deployment, and operation of PPC solutions burdensome and complex. Users and researchers often cite issues like prohibitive computational overhead (especially in monolithic universal solutions), reliance on exotic tech stacks and dependencies, the lack of cryptography skills among developers, and the lack of understanding and support from data protection officers.

In this study, we introduce an architecture that addresses several perceived "pain points" and offers a simple yet effective framework for employing various PPC technologies. The objective is to separate the complexity of cryptographic protocols from business logic concerns. This minimalist solution aims to be usable by small teams without a cryptography background. It should support multiple topologies between data owners, processing nodes, and data consumers, allowing developers to use any language or libraries. The data flow should be transparent and easy to secure.

Ultimately, the solution must possess a level of efficiency that allows it to function seamlessly in Internet of Things (IoT) environments and have the capacity to expand to accommodate potentially extensive networks.

MATERIALS AND METHODS

Challenges and Objectives

The subsequent challenges have consistently surfaced in literature, emerged during discussions with potential early adopters, or have been directly encountered by the authors. Each challenge has prompted the establishment of a distinct design objective aimed at mitigating these concerns (refer to Table 1).

Table 1. Challenges and Corresponding Design Objectives

Pain Points	Design Goals
related to capabilities (C)	
front-end and business logic developers rarely have any expert knowledge of PPC	provide high-level computing routines that hide low-level cryptography protocols **(C1)**
PPC is inaccessible to marginal groups lacking computing and personal resources	build a minimalistic solution that can be run by a single developer on a Raspberry Pi [19] **(C2)**
PPC is difficult to teach and experience in the limited time frame of a typical lesson	provide a propaedeutic solution that works in a school or university teaching setting **(C3)**
related to development (D)	
PPC often appears as the core functionality, so far as to even require to be a main routine	PPC should be a network-level concern, separated from high-level concerns **(D1)**
introducing PPC functionality to a business logic often requires a complete rework	enable piecewise introduction of PPC into an existing legacy business logic codebase **(D2)**
PPC frameworks require a specific tech stack and introduce a lot of dependencies	client side and the core of the server side should be free of any dependencies **(D3)**
coding for any particular PPC framework locks the developer to a specific language	let client-side developers freely choose their language or keep the legacy one **(D4)**
related to security concerns (S)	
some PPC frameworks are developed by non-experts in cryptography and are unsafe	do not reinvent the wheel; make use of existing and proven PPC frameworks **(S1)**
PPC involves sensitive data that should not be visible to the front end or the outside	enable topologies with data flow confined to trusted machines on the backend **(S2)**
many PPC use cases involve a third-party researcher who must be able to run analyses	enable topologies with control flow coming from an external researcher **(S3)**
every PPC calculation requires a complete re-evaluation by data security officers	separate topology, protocol, and function so they can be assessed independently **(S4)**
related to deployment and operation (O)	
PPC often requires all parties to agree on the exact same tech stack and IT environment	enable joint computation between parties using different hardware or software **(O1)**
without a lot of experience, it is often unclear which PPC framework is best suited for a task	frameworks should be replaceable without the need to rewrite business logic **(O2)**
universal PPC solutions often have enormous overhead in terms of space and processing	provide "small and fast" solutions that cater to the most often encountered tasks **(O3)**

Derived Architecture

At its core, the architecture is guided by a dual separation: Firstly, the necessity to disentangle business logic from cryptography protocols (enforcing a separation of concerns and dependency inversion). Secondly, the imperative to segregate data flows from distinct data owners until they interact with the underlying cryptography layer (prioritizing privacy). This dual separation is effectively addressed through a client/server architecture. The provision of server-side Privacy-Preserving Computation (PPC) protocols via microservices and a web API to the client-side business logic aligns with several of the aforementioned design goals.

However, for clients to remain entirely unburdened by cryptography concerns, they must refrain from transmitting cryptographic shares to the server and should instead have the capability to send raw data in plain text. This is crucial because the client should remain agnostic regarding the specific PPC

protocol executed by the server, considering that different protocols necessitate distinct generations of shares. Opting for minimalistic and streamlined clients, prioritizing simplicity on the client side, we make the unconventional decision that each data owner operates their own server. In the language of Privacy-Preserving Computation (PPC), this translates to having as many compute nodes as there are input/data nodes, with the possibility of them coinciding. The resultant design choices are detailed in Table 2.

Table 2. Decisions in Design and the Underlying Rationale

Design Decision	Rationale and Addressed Design Goals
client–server architecture	• offload computationally expensive cryptography to the server **(D1)** • separate business logic concerns (client-side) from cryptography concerns (server side) **(C1, D1, O2)**
one dedicated server per client	• allows clients to remain lightweight and generalist API wrappers without specific encryption logic **(D3)** • servers can be trusted by their own client/data owner **(S2, S3)** • for less attack surface, it is possible to keep data entirely server side with only control flow coming from the client **(S2, S3)** • easy to analyze and straightforward to secure **(S4, S2)** • easy to explain and understand in a propaedeutic setting **(C3)**
solution should be a middleware	• encapsulate low-level cryptographic scripts and provide them to the client as high-level macros **(C1, D1)** • provide the same macros for different PPC backends **(O2, S4)** • hosting one or several established PPC frameworks **(S1)**
provide microservices	• rebuild business logic piecewise in a privacy-friendly fashion **(D2)** • provide highly optimized microservices for specific business problems instead of slow universal monoliths **(O3, C2)**
provide RESTful API	• easy to provide API wrapper in any programming language **(D4)** • client remains free of any PPC-specific dependencies **(D3)**
represent server-side objects 1:1 client-side	• make client-side code easy to read and write **(D1, C1, C3)** • allow the client to remain lightweight yet powerful **(D3, C2, C3)** • allow server-side-only data flow **(S2, S3, S4)**
implement server middleware in Python (optional)	• core middleware can be implemented in pure Python without any additional dependencies except for a webserver **(D3)** • Python is available on most any platform **(O1, C2)** • Python is popular in propaedeutic settings and data science **(C3)**

Conceptual Framework of Federated Secure Computing

Federated Secure Computing establishes connections among diverse systems (federation) through privacy-preserving computing protocols (secure computing), as illustrated in Figure 1.

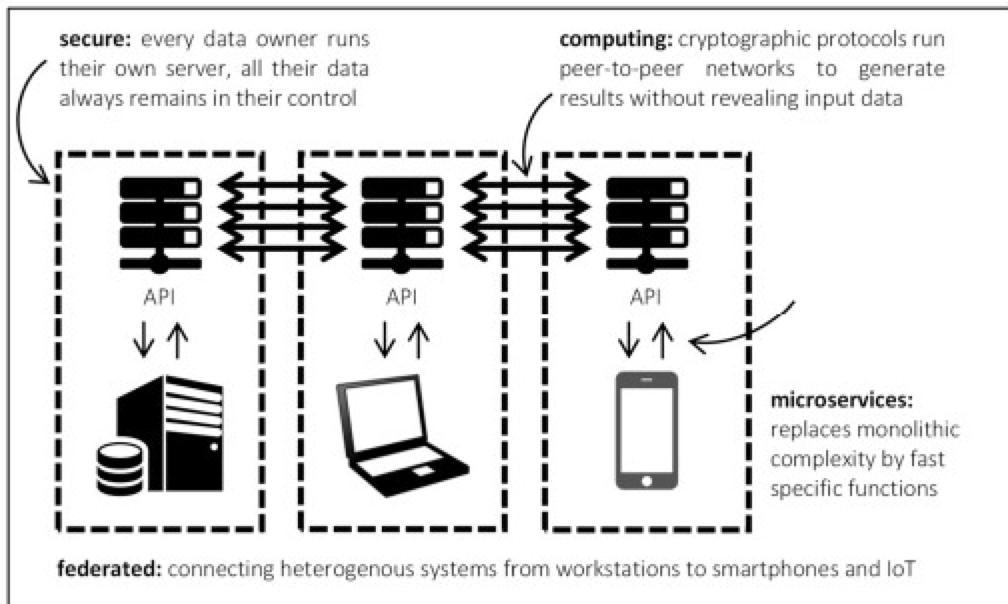


Figure 1. Federated Secure Computing

Functioning as middleware between client-side business logic and server-side cryptography backends, it encapsulates secure computing functionality via specific microservices. These functionalities are then exposed to clients through an easily accessible API, as depicted in Figure 2.

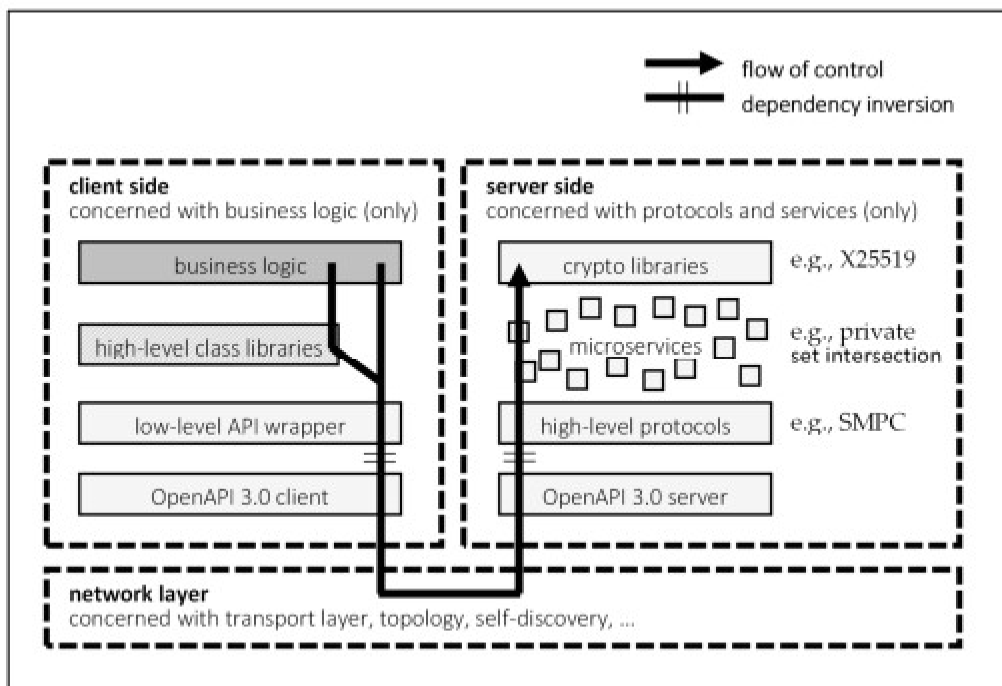


Figure 2. Flow of control, separation of concerns between client and server side, and API/microservice architecture

Client–Server Topologies

While the insistence on having precisely one server per data owner might seem robust, there are no actual constraints on the topology of the privacy-preserving computing (PPC) protocols utilized in the backend.

In PPC terminology, there are data nodes (providing input data), compute nodes (executing the protocols), and researcher nodes (managing control flow and receiving results). In our context, these correspond to server nodes and client nodes, which may or may not align with data, compute, and/or researcher nodes.

Example 1: Clients as Data and Researcher Nodes, and Servers as Compute Nodes

This topology is suitable when multiple equal and concurrently active researchers operate within a symmetric peer-to-peer network.

Clients transmit unencrypted input data and control flow to their respective servers. The servers host the PPC protocol, acting as compute nodes. They disassemble the input data into cryptographic shares, inject it into the protocol, execute the protocol on encrypted shares, and transmit the results back to their respective clients, as illustrated in Figure 3.

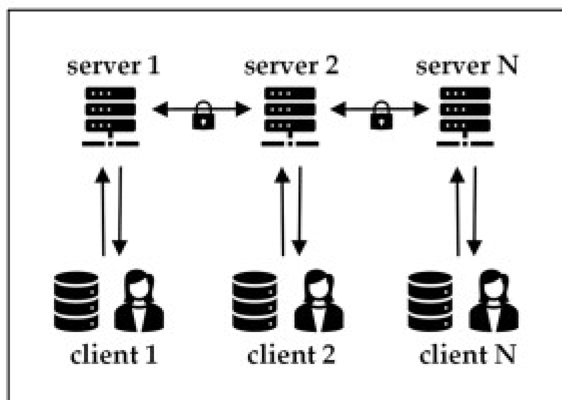


Figure 3. Peer-to-Peer Topology with Symmetric Structure, Featuring One Client (Researcher) for Each Server

This topology is exemplified by the propaedeutic protocol Simon (Simple Multiparty computation) and is presented as a functional illustration in the reference implementation.

Example 2: Servers Function as Data and Compute Nodes, with a Single Client Operating as the Researcher Node

In contrast to Example 1, data is stored on the server rather than the client. This scenario is more likely in institutions where data should remain confidential even from their own clients and researchers. Here, there's no necessity for more than one researcher, although having one researcher per server remains a viable option. The lone researcher can dispatch control flow to all servers, making synchronization straightforward, and solely receive the computation's outcome (without accessing input data on the servers), as depicted in Figure 4.

When employing server-side object representation, it becomes feasible to create wrappers for server-side database handles, enabling their access through the client. However, caution must be exercised to securely protect the API, particularly through object-level authorization.

This topology proves advantageous when there is a privileged researcher and several independent contributors, such as a university hospital researching data from teaching hospitals, a government agency utilizing data from regional bodies, a parent company analyzing subsidiaries, or an industry association offering benchmarks to member companies.

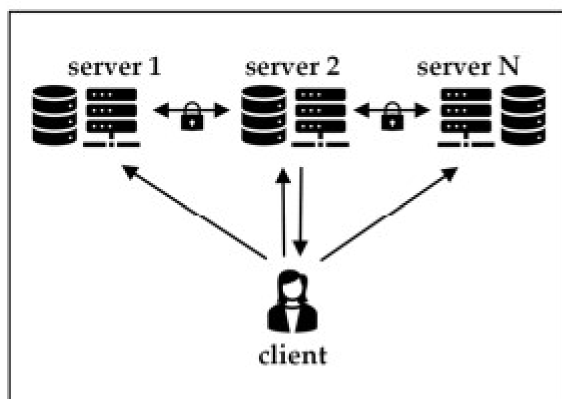


Figure 4. Topology Featuring a Sole Central Client (Researcher)

Example 3: Servers Exclusively Host Middleware, with Additional Compute Nodes in the Backend

Certain privacy-preserving computing (PPC) protocols may necessitate a dedicated compute cluster. For instance, some Secure Multiparty Computation (SMPC) protocols utilize three distinct compute nodes, regardless of the quantity of data nodes. In this scenario, the role of the Federated Secure Computing servers is essentially to serve as gateways, hosting a translatory middleware, as illustrated in Figure 5.

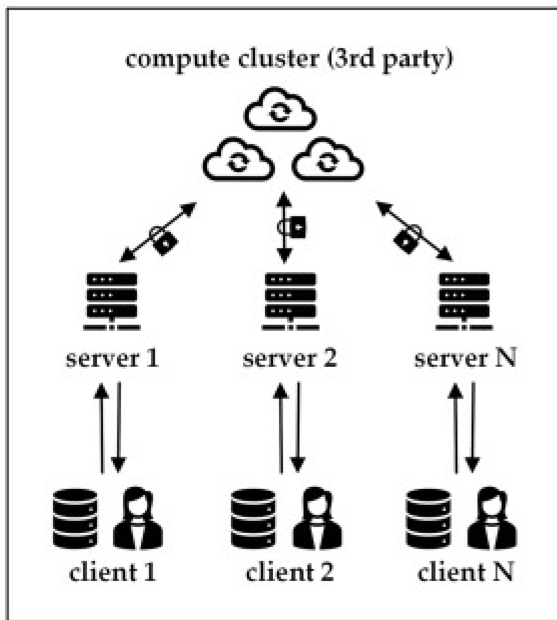


Figure 5. Configuration with Servers Solely Running Middleware and a Compute Cloud in the Backend

In this setup, servers exclusively operate middleware, with the compute cloud serving as the backend. The middleware takes in input data, translating them into cryptographic shares based on the compute cluster's protocol. Additionally, the middleware receives control flow instructions from the client and communicates accordingly with the compute cluster.

This topology proves advantageous when seeking to integrate the straightforward client-side approach of Federated Secure Computing with a more sophisticated and comprehensive solution for executing actual computations on the backend. For instance, a Carbyne Stack (Robert Bosch GmbH, 2022, Stuttgart, Germany) compute cluster could serve as a valuable backend.

Example 4: DataSHIELD

DataSHIELD [20,21] stands as a widely used privacy-preserving computing (PPC) solution in academic and data science contexts. It involves a central compute node that receives aggregated data from various data nodes, further consolidates it, and subsequently transmits the summary statistics to the researcher.

Should there be a desire to encapsulate the DataSHIELD server within a middleware provided by Federated Secure Computing, the depicted topology would resemble Figure 6.

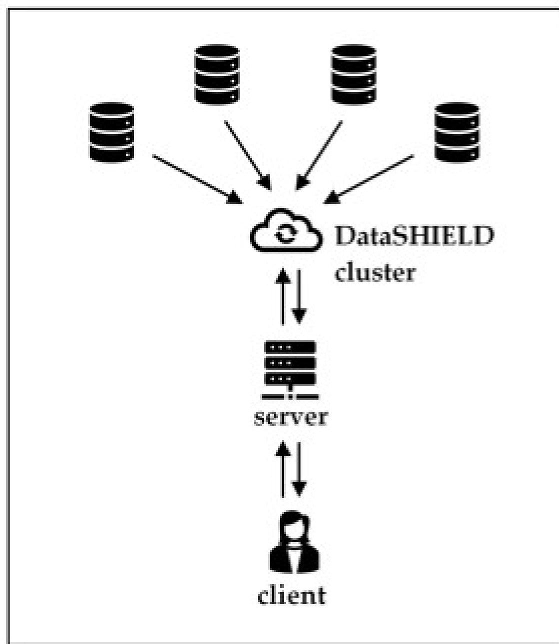


Figure 6. Topology of DataSHIELD

In a sense, this represents a fusion of Example 2 (involving a single researcher) and Example 3 (emphasizing the pure middleware functionality of Federated Secure Computing). This configuration proves beneficial if there is a preference for using client-side languages other than R for script development. Alternatively, it might be advantageous to incorporate DataSHIELD for its statistical capabilities, particularly when processing metadata of data analyzed comprehensively by other protocols like secure multiparty computation.

Heterogeneous Networks

The architecture is versatile enough to accommodate diverse topologies. Referring to Example 1, certain sites may store their data on the client side (as illustrated in Figure 3), while others house their databases on the server side (as depicted in Figure 4). In such cases, the sites merely require distinct data input instructions in their respective client-side code. In Examples 1 and 2, there could be instances where certain sites have a dedicated researcher (as shown in Figure 3), while others form—possibly multiple—pools overseen by a central researcher. Layered topologies are also viable, where the output of one federated computation layer becomes the input for the next layer. The primary requirement is that all parties within one layer or federated cloud agree on the same cryptographic protocol for server-side execution. Moreover, a scenario could arise where some parties use

middleware, while others execute their frameworks natively, as long as cryptographic communication remains compatible.

Client-Side Stack Representation of Server-Side Objects

Our overarching design objective is to simplify client-side business logic development. We aim to eliminate specific dependencies on the client side and ensure seamless interaction through the API. Consequently, the goal is to write client-side code with the utmost transparency, exemplified as follows (see Listing 1):

Listing 1. Illustration of the Desired Interaction Between Client-Side Code and Server-Side Objects

```
import federatedsecure.client

# connect to the server, return API handle
api = federatedsecure.client.connect(“https://my.server”)

# find a microservice that matches some requirements
microservice = api.create(functionality = “can do some stuff”})

# connect to some specific server-side database
database = api.create(connector = “myconnector”, version = “1.2.3”)

# fetch input data
data = database.get_handle().query(row = 2, column = 5)

# do some server-side computation
result = microservice.compute(data)

# download and output the result
print(api.download(result))
```

There are two functions facilitating communication between the server side and the client side:

- api.create returns a handle to a top-level server-side object, typically representing a microservice. Various arguments are supplied to describe the desired microservice.
- api.download serializes the server-side data associated with a particular handle and transmits them back to the client.

This implies that all other variables in the pseudocode above (microservice, database, data, and result) serve as mere handles for the corresponding server-side objects. The client can seamlessly access these handles through the API, triggering server-side actions without any dependencies on the client side.

To accomplish this, we employ a wrapper class named "Representation." As its name suggests, it serves as a representation of server-side objects. The class stores a pointer to the API (enabling the triggering of API requests) and a unique identifier (UUID) corresponding to the server-side object. Access to member variables and functions is then mirrored on the API by passing the UUID.

In Python, this process is particularly straightforward (refer to Listing 2):

Listing 2. Simplified Representation Class

```
Class Representation:

def __init__(self, api, uuid):
self.api = api
self.uuid = uuid

def __getattr__(self, member_name):
return self.api.attribute(self.uuid, member_name)

def __call__(self, *args, **kwargs):
return self.api.call(self.uuid, *args, **kwargs)
```

Consider the scenario where database.get_handle().query(row=2, column=5) is executed. Surprisingly, this single line actually generates four (!) representations: (1) of the member function get_handle, (2) of the outcome after invoking that member function without arguments, (3) of the query function associated with that result, and (4) of the outcome after executing the query function with specified arguments.

It's worth noting that not all programming languages provide such convenient syntactic sugar. For instance, in R, the equivalent code would appear as follows (refer to Listing 3):

INFOTECH
Journal of Applied Sciences, Informatics, Computing and Knowledge Society
Volume 3, No. 1, January/December, 2022

Listing 3. In certain programming languages, the code on the client side may exhibit more verbosity compared to Python.

```
source ("../federatedsecure/client.r")

# connect to the server, return API handle
api <- Api("https://my.server")

# find a microservice that matches some requirements
microservice <- api$create(kwargs = list(
functionality = "can do some stuff"))

# connect to some specific server-side database
database <- api$create(kwargs = list(
connector = "myconnector", version = "1.2.3"))

# fetch input data
func_handle <- database$attribute("get_handle")
handle <- func_handle$call()
func_query <- handle$attribute("query")
data <- func_query$call(list(row = 2, column = 5))

# do some server-side computation
func_compute <- microservice$attribute("compute")
result <- func_compute$call(list(data = data))

# download and output the result
print(api$download(result))
```

API Wrapper

Utilizing the Representation approach allows channeling the complete API traffic through a minimal number of RESTful endpoints (refer to Table 3).

Table 3. API Endpoints

Verb	Endpoint	Server-Side Effect and Response
GET	/representations	• list of top-level microservices
POST	/representations	• finds matching top-level microservice • returns uuid representing the microservice
PUT	/representations	• upload data and store them on the server side • returns uuid representing the data
PATCH	/representation/{uuid}	• calls server-side function represented by uuid • stores the return value on the server side • returns uuid representing the return value
GET	/representation/{uuid}/{attr}	• gets attribute (e.g., child variable, member function) of object represented by uuid • stores the pointer on the server side • returns uuid representing the attribute
GET	/representation/{uuid}	• serializes the object represented by uuid • returns the serialized data
DELETE	/representation/{uuid}	• deletes the object represented by uuid

In the context of these specifications (as outlined in Listing 4), the implementation of api.create, api.download, api.call, and api.download—as utilized in the main routine and the Representation class above—is straightforward.

Listing 4. Simplified Api Class

```
class Api:

def __init__(self, url):
self.http = HttpInterface(url)

def list(self):
return self.http.GET('representations')

def create(self, *args, **kwargs):
response = self.http.POST('representations',
body = {'args':  args, 'kwargs':  kwargs})
return Representation(self, response['uuid'])

def upload(self, *args, **kwargs):
response = self.http.PUT('representations',
body = {'args':  args, 'kwargs':  kwargs})
return Representation(self, response['uuid'])

def call(self, uuid, *args, **kwargs):
response = self.http.PATCH('representation', uuid,
body = {'args':  args, 'kwargs':  kwargs})
return Representation(self, response['uuid'])

def attribute(self, uuid, attr):
response = self.http.GET('representation', uuid, attr)
return Representation(self, response['uuid'])

def download(self, representation):
response = self.http.GET('representation',
representation.uuid)
return response['object']

def release(self, uuid):
self.http.DELETE('representation', uuid)
return None
```

Client Design Considerations

Programming Language: Currently, API wrappers are available in Python, R, and JavaScript. Given the lightweight nature of the client, containing only the classes Api and Representation along with an HTTP interface, developing API wrappers in other languages is straightforward. The client's thin structure facilitates easy adaptability to various programming languages, and even a tool like curl can suffice.

Thin Client: It is essential to maintain a thin client, allowing flexibility for client-side developers to choose their preferred language. Keeping the client small (i.e., kilobytes) ensures suitability for IoT settings.

Macros on the Server Side: Macro functionalities should primarily reside on the server side. For instance, combining multiple steps into a single line of client code, such as connecting to a database, obtaining a handle, reading data, and storing it, should be implemented as a small server-side extension. This approach ensures that the functionality is universally available to all clients, eliminating potential slowdowns and heavy payload issues associated with multiple API calls.

Securing the API: While primarily a server-side concern, authentication and authorization considerations also extend to the client side. Federated Secure Computing is designed with an emphasis on educational settings, but if deployed in production, security adaptations should align with the organization's specific implementation.

Full RPC Framework: The provided implementation is minimalistic and tailored for educational purposes. In a production environment, it might be prudent to leverage a more comprehensive and stable framework for remote procedure calls.

Server-Side Stack

Registry, Discovery, and Bus: The core middleware comprises a registry of server-side objects, a bus for accessing them, and a discovery mechanism for registering top-level objects. The registry manages pairs of top-level microservices and their descriptions, offering functionality to register, list, or fetch specific microservices. The abstraction of descriptions ensures decoupling between the registry and microservices (following the dependency–inversion principle). The discovery mechanism identifies available microservices during startup, enabling them to register without modifying the server (in line with the open–closed principle). The bus facilitates communication between server-side objects, encompassing both microservices and dynamic instances created at runtime.

OpenAPI 3.0: The API is defined by an OpenAPI 3.0 [22] compliant description. For example, the PATCH endpoint is articulated as follows (refer to Listing 5).

Listing 5. Excerpt from the OpenAPI 3.0 Specification

```
 /representation/{representation_uuid}:
patch:
summary:  call a server-side object
description:  call a server-side object such as a static
function, a member function, or in case of a class, its
constructor
operationId:  call_representation
parameters:
- in:  path
name:  representation_uuid
required:  true
schema:
type:  string
format:  uuid
requestBody:
$ref:  '#/components/requestBodies/ArgsKwargs'
responses:
'200':
$ref:  '#/components/responses/ResponseOk'
'default':
$ref:  '#/components/responses/ResponseError'
```

Representing Server-Side Objects

At its core, the bus provides essential functionalities, including the creation of representations for registered microservices, direct creation of representations for uploaded data, generation of representations for member variables and functions within represented objects, invocation of a represented function, downloading the content associated with a representation, and releasing a representation.

Figure 7 for a visual depiction of the life cycle of representing server-side objects

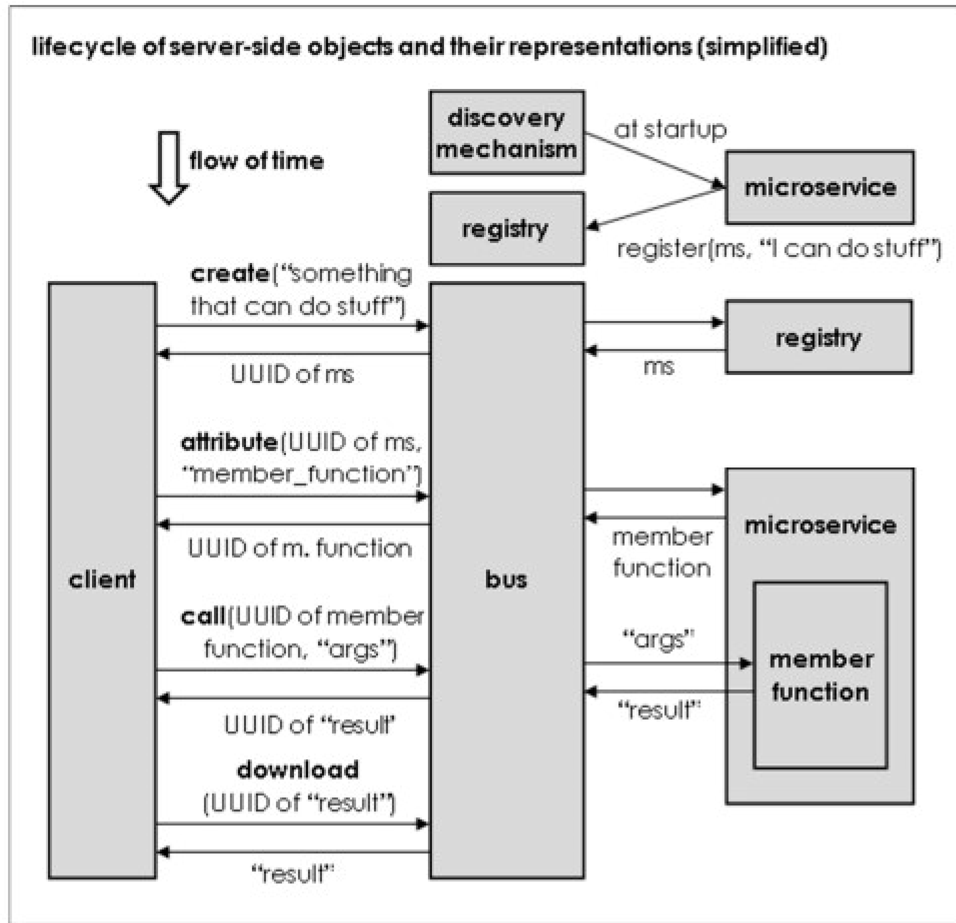


Figure 7. Lifecycle of Server-Side Objects and Their Representations (Simplified).

The life cycle initiates with microservices announcing their presence and functionalities to the registry during startup. Upon client requests for a specific microservice with particular capabilities, a handle is stored on the bus, and a corresponding UUID is provided to the client.

Subsequent interactions involve the client referencing the microservice through the assigned UUID, allowing requests for attributes or member functions. This process iterates if needed. When a handle represents a callable function, the client can invoke it with additional arguments. The result is stored on the bus, and a new UUID is returned to the client.

A basic illustration of this functionality is outlined in Listing 6. Upon reaching the computation result, the client can request the actual download of the result. It's crucial to address security concerns by

limiting download access to designated output objects, necessitating proper object-level authorization in production settings.

Ultimately, the client has the option to release any unnecessary representations, leading to the potential deletion of temporarily stored objects or the discarding of static microservice representations on the bus.

Listing 6. Simplified Implementation of Server-Side Object Calls

```
def call_representation(self, representation_uuid, body):

args, kwargs = self.get_arguments(body)
pointer = self.lut_uuid_to_repr[representation_uuid]
result = pointer(*args, **kwargs)

if result is None:
return None

uuid = str(uuid.uuid4())
self.lut_uuid_to_repr[uuid] = result
return uuid
```

Best Practices

- Ensure the client sends appropriate delete requests to the server when client-side representations are discarded or go out of scope to prevent a buildup of obsolete representations on the bus and potential memory leakage on the server.
- Implement additional garbage collection mechanisms, such as automatic removal of unused representations after a specified grace period, considering the server's inability to control the termination of client-side scripts.

Additional Best Practices

- Maintain look-up tables for commonly used objects on the server to avoid issuing new UUIDs every time.
- Balance the overhead of creating representations with microservice design, considering that few objects with many properties/methods require fewer representations. Deeply nested object hierarchies may generate more intermediate handles.

Microservices

- The architecture is flexible regarding the types of microservices hosted.
- In the context of privacy-preserving computation (PPC), essential microservices include one or more PPC protocols for building peer-to-peer networks, accepting input data, generating cryptographic shares, and executing PPC protocols. They may interact with peers through the bus, API, or their own networks.
- Basic microservices for synchronization may be required, facilitating tasks like broadcasting public parameters or controlling joint computation flow through semaphores and signals.
- Optional helper microservices can aid server-side integration into non-cryptographic infrastructure, providing interfaces to database prompts or IoT data acquisition.

Webserver and API

- The server will expose the bus's public functionality to the client through an API.
- A standard webserver will be necessary for this purpose.
- In a production environment, it is imperative to adhere to standard security practices for securing both the webserver and API. This includes user authentication and ensuring proper object-level authorization.

Programming Language

While there is no specific programming language requirement for implementing a Federated Secure Computing server, the provided propaedeutic reference implementation is in Python. However, different microservices can be implemented in various languages, communicating through the API as needed.

Results: Implementation

Namespaces: The following namespaces are either currently in use or reserved for future utilization (refer to Table 4).

Table 4. Namespaces

Language	Namespace Structure
Python	• federatedsecure • federatedsecure.client • federatedsecure.server • federatedsecure.services • federatedsecure.services.<name>.* (see below) • federatedsecure.services.<category>.<name>.* (see below)
Java	• com.federatedsecure.*

Packages

A variety of packages are accessible, as outlined in Table 5.

Table 5. Packages

Repository	Packages
PyPI	• federatedsecure-client • federatedsecure-server • federatedsecure-simon

Repository Structure

As of the current writing, the GitHub repositories detailed in Table 6 are publicly available, providing insight into the structure and contents of the implementation.

Table 6. GitHub Repository Structure and Contents

Repository	Contents
github	• top-level README.md of the organization
api	• OpenAPI 3.0 specification used by both client and server
client-<language>	• client libraries providing API wrappers in multiple languages • top-level directory may contain "src", "test", "docs", etc. • examples beyond a simple hello world should go with the • services' repositories and should work with multiple clients

Repository	Contents
server	• core middleware as importable library (without webserver runtime) • implemented in Python only (no top-level language directories) • top-level directory contains "src", "docs", "examples", and "pypi" • the "pypi" directory contains the Python Package Index manifest
service-<name>	• larger, complex microservices • typically, PPC protocols and interfaces to 3rd party PPC backends • e.g., "service-simon" (SImple Multiparty computatiON) is a simple, propaedeutic secure multiparty computation (SMPC) protocol • e.g., "service-datashield" would be an interface to DataSHIELD
utility-<category>-<name>	• smaller, helper microservices • e.g., "utility-database-mysql" could expose mysql.connector from the mysqlclient package by wrapping it into a small microservice
webserver-<name>	• premade webserver runtimes • e.g., "webserver-connexion" or "webserver-django" • further webserver stubs can be generated • by the Swagger utilities from the API definition
whitepaper	• this whitepaper

Correspondence

The connection between namespaces, PyPI packages, and GitHub repositories is delineated in Table 7.

Table 7. Relation between namespaces, packages, and GitHub repositories

Python Namespace	PyPI Package	GitHub Repository
federatedsecure.client	federatedsecure-client	client-python
federatedsecure.server	federatedsecure-server	server
federatedsecure.services.<name>	federatedsecure-<name>	service-<name>
federatedsecure.services.<category>.<name>	federatedsecure-<category>-<name>	utility-<category>-<name>

Code Availability and Licensing

The software is freely accessible and open-source through https://github.com/federatedsecure, retrieved on August 23, 2023.

All public repositories within this GitHub organization are equipped with the MIT license.

Installation

Server-Side Installation

To begin, set up a web server for hosting the API. Flask combined with Connexion offers a minimalist choice with minimal overhead. Alternatively, Django provides a more comprehensive option. Both are supplied as pre-made templates.

```
git clone https://github.com/federatedsecure/webserver-connexion
cd webserver-connexion
pip install -r requirements.txt
```

Next, install the code server middleware and any additional protocols you might want to use, e.g.,

```
pip install federatedsecure-server
pip install federatedsecure-simon
```

The middleware will automatically discover the plugin, so there is no more additional setup. Run the server by, e.g.,

```
python ./src/__main__.py --port = 55500
```

You can check that the server is running by browsing to

```
curl http://localhost:55500/representations
```

3.2.2. Client-Side Installation

Client-side installation is likely a single command:

```
pip install federatedsecure-client
```

On machines where pip is not readily available as a command line tool (e.g., Android), the following workaround works directly in Python:

```
import pip
pip.main([‘install’, ‘federatedsecure-client’])
```

Benchmarking

The ensuing benchmarks provide insights into the anticipated performance of a secure multiparty computation (SMPC) backend utilizing task-specific microservices. For this purpose, we employ the introductory Python package "Simon" (SImple Multiparty computatiON). Generally, monolithic universal SMPC solutions, such as garbled circuits, might exhibit slower performance, while optimized implementations in compiled languages tend to be faster. It is crucial to note that alternative methods, like fully homomorphic encryption or trusted computing, as seen in DataSHIELD, could

present distinct performance characteristics. The goal here is not an exhaustive analysis of the performance of various methods and algorithms, as this has been extensively covered in the literature. Instead, this serves as evidence that basic calculations for average-sized problems can be executed with minimal overhead using the presented middleware.

Impact of Server Hardware

The primary computational load is shouldered by the server side. In the subsequent benchmark, two or three servers and two or three clients concurrently operate on the same local host machine (refer to Table 8). The workstation showcased the capability to complete tasks in approximately one second on average. Meanwhile, the laptop took about two to three times longer due to lower CPU and RAM clocks, energy conservation mode, and concurrent engagement in typical office tasks. Remarkably, the Raspberry Zero, priced at five USD, demonstrated the ability to concurrently run three Federated Secure Computing servers and clients. Although the BCM2835-based system-on-a-chip operates at an order of magnitude slower pace than larger machines, it may still prove valuable in propaedeutic or Internet of Things applications.

Table 8. Benchmark Results for Speed Based on Server Hardware (in seconds).

Task	Workstation [1]	Laptop [2]	Raspberry Zero [3]
horizontally partitioned data (without record linkage)			
floating point additions [4]	0.10 ± 0.01	0.26 ± 0.01	6.8 ± 1.3
matrix multiplications [6]	0.26 ± 0.02	0.64 ± 0.24	7.7 ± 0.2
histograms [5]	0.25 ± 0.04	0.59 ± 0.10	16.4 ± 0.3
contingency tables [5]	0.38 ± 0.07	1.00 ± 0.12	27.8 ± 0.6
univariate statistics [5]	0.64 ± 0.05	1.71 ± 0.18	52.4 ± 0.5
bivariate statistics [5]	1.93 ± 0.05	5.70 ± 0.11	155.7 ± 1.7
set intersections [5]	0.57 ± 0.06	1.30 ± 0.07	35.7 ± 0.5
set intersection size [5]	0.48 ± 0.08	1.18 ± 0.10	35.9 ± 0.7
vertically partitioned data (with record linkage)			
contingency tables [5]	1.33 ± 0.16	3.29 ± 0.35	84.3 ± 2.1
OLS regression [6]	0.86 ± 0.01	0.24 ± 0.01	5.8 ± 0.2

[1] Intel Core i7-9700K, 3.6 GHz, 96 GB DDR4-3600. [2] Intel Core i5-6200U, 2.3 GHz, 8 GB DDR4-2133. [3] Broadcom BCM2835, 1.0 GHz, 512 MB LPDDR2-SDRAM. [4] M = 3 parties; N = 100 data samples each. [5] M = 2 parties; N = 100 data samples each. [6] M = 2 parties; N = 100 elements in 10×10 matrix.

Impact of Server-to-Server Connectivity

Many secure multiparty computation protocols involve multiple rounds of communication between servers, making network overhead a crucial factor affecting computing time.

The following benchmark examines the connection of two servers through different means, introducing varying network latency. The baseline involves a workstation, as mentioned earlier, hosting both servers. The internal latency is well below 1 millisecond, essentially zero. In the second scenario, the workstation and the laptop, as mentioned earlier, are connected by ethernet cables and WLAN through a router, each with 2 milliseconds latency. In the third scenario, another fast server is linked to the workstation over the public internet, introducing 28 milliseconds of latency. Refer to Table 9.

Table 9. Speed benchmarks based on server connectivity (seconds).

Task	Localhost [1] (<1 ms ping)	LAN/WLAN [2] (2 ms ping)	Internet [3] (28 ms ping)
horizontally partitioned data (without record linkage)			
floating point additions [4]	n/a	0.50 ± 0.02	2.5 ± 0.7
matrix multiplications [5]	0.26 ± 0.02	0.81 ± 0.22	2.7 ± 0.8
histograms [4]	0.25 ± 0.04	1.72 ± 0.05	9.4 ± 2.0
contingency tables [4]	0.38 ± 0.07	3.12 ± 0.13	16.1 ± 4.2
univariate statistics [4]	0.64 ± 0.05	4.10 ± 0.74	22.3 ± 4.5
bivariate statistics [4]	1.93 ± 0.05	11.14 ± 0.54	59.2 ± 5.3
set intersections [4]	0.57 ± 0.06	1.50 ± 0.27	4.0 ± 0.6
set intersection size [4]	0.48 ± 0.08	1.30 ± 0.06	3.1 ± 0.5
vertically partitioned data (with record linkage)			
contingency tables [4]	1.33 ± 0.16	4.78 ± 0.25	14.9 ± 1.9
OLS regression [5]	0.86 ± 0.01	0.52 ± 0.07	2.3 ± 0.4

[1] 1× Intel Core i7-9700K, 3.6 GHz, DDR4-3600, running both servers. [2] 1× Intel Core i7-9700K, 3.6 GHz, DDR4-3600, and 1× Intel Core i5-6200U, 2.3 GHz, DDR4-2133, connected by LAN router, 2 ms RTD. [3] 1× Intel Core i7-9700K, 3.6 GHz, DDR4-3600, and 1× Intel Xeon Silver 4310, 2.1GHz, DDR4-2666; connected by internet, 28 ± 1 ms RTD. [4] M = 2 parties; N = 100 data samples each. [5] M = 2 parties; N = 100 elements in 10×10 matrix.

In the WLAN configuration, networking overhead roughly doubles the overall computing time. In the internet setting, networking overhead increases computing time about fivefold. Hence, in practical scenarios, placing servers of different parties close to each other, such as in the same physical data center or hosting them on a common cloud infrastructure, will be beneficial.

Impact of Client–Server Connectivity

In the final speed benchmark, the servers operate on the same machine as before, but the clients connect through different means. In the baseline, the clients run on the same physical machine as mentioned earlier. In the second case, the clients run on a separate laptop, connected by LAN/WLAN to the workstation as before. In the third setting, the clients run on smartphones, connecting to the

workstation through public mobile internet services. Clients connected through localhost and LAN/WLAN client were about as fast, but connections over the mobile network were slower by an order of magnitude due to increased round-trip delays (refer to Table 10).

Table 10. Speed benchmarks based on client–server connectivity (seconds).

Task	Localhost [1]	LAN/WLAN [2]	Mobile Network [3]
horizontally partitioned data (without record linkage)			
floating point additions [4]	0.10 ± 0.01	0.26 ± 0.01	6.8 ± 1.3
matrix multiplications [6]	0.26 ± 0.02	0.64 ± 0.24	7.7 ± 0.2
histograms [5]	0.25 ± 0.04	0.59 ± 0.10	16.4 ± 0.3
contingency tables [5]	0.38 ± 0.07	1.00 ± 0.12	27.8 ± 0.6
univariate statistics [5]	0.64 ± 0.05	1.71 ± 0.18	52.4 ± 0.5
bivariate statistics [5]	1.93 ± 0.05	5.70 ± 0.11	155.7 ± 1.7
set intersections [5]	0.57 ± 0.06	1.30 ± 0.07	35.7 ± 0.5
set intersection size [5]	0.48 ± 0.08	0.51 ± 0.14	1.62 ± 0.20
vertically partitioned data (with record linkage)			
contingency tables [5]	1.33 ± 0.16	3.29 ± 0.35	84.3 ± 2.1
OLS regression [6]	0.86 ± 0.01	0.24 ± 0.01	5.8 ± 0.2

[1] servers and clients are running in separate CPU cores on same machine. [2] clients on laptop connected through Intel Dualband-Wireless-AC 8260. [3] clients on three separate mobile phones (2× Samsung SM-G52F/DS "XCover 5") connecting through 4G network of Telefónica S.A. in Germany. [4] M = 3 parties; N = 100 data samples each. [5] M = 2 parties; N = 100 data samples each. [6] M = 2 parties; N = 100 elements in 10 × 10 matrix.

Code Size Benchmarks

The API wrapper on the client side and the middleware stub on the server side are both compact, as indicated in Tables 11 and 12.

Table 11. Dimensions of the client-side API wrapper

Language	Without HTTP Interface	With HTTP Interface
Python	2.8 kilobyte	7.4 kilobyte
R	2.1 kilobyte	6.6 kilobyte
Javascript	2.1 kilobyte	3.4 kilobyte

Table 12. Dimensions of the server-side API wrapper

Language	Without Webserver	With Webserver
Python	12.4 kilobyte	31.6 kilobyte

Setting up in IoT environments was uncomplicated on Raspberry Pi and Raspberry Pi Zero, thanks to the availability of regular and dedicated Linux distributions. Our testing involved Ubuntu LTS 20.04 on Raspberry Pi 4 Model B and Raspberry Pi OS on Raspberry Pi Zero. Both distributions feature an apt package manager, allowing easy installation of Python and pip in the usual manner.

For Android smartphones, installation requires a slight workaround, as apt and pip are typically unavailable. We utilized QPython3, a free and open-source software, which can be installed via an Android package (APK) obtained from GitHub or traditional app stores. In this case, instead of using pip directly in the command line, importing pip as a Python module is necessary, and it can then be employed to install the required packages.

```
import pip
pip.main([‘install’, ‘federatedsecure-client’])
```

Following this approximately five-minute procedure, it becomes possible to conveniently develop Federated Secure Computing applications on Android (refer to Figure 8). In a production environment, developers may opt to create applications on a workstation (e.g., via USB) and then distribute the packaged app. Nevertheless, for educational settings, the capability to develop applications directly on smartphones is a valuable feature.

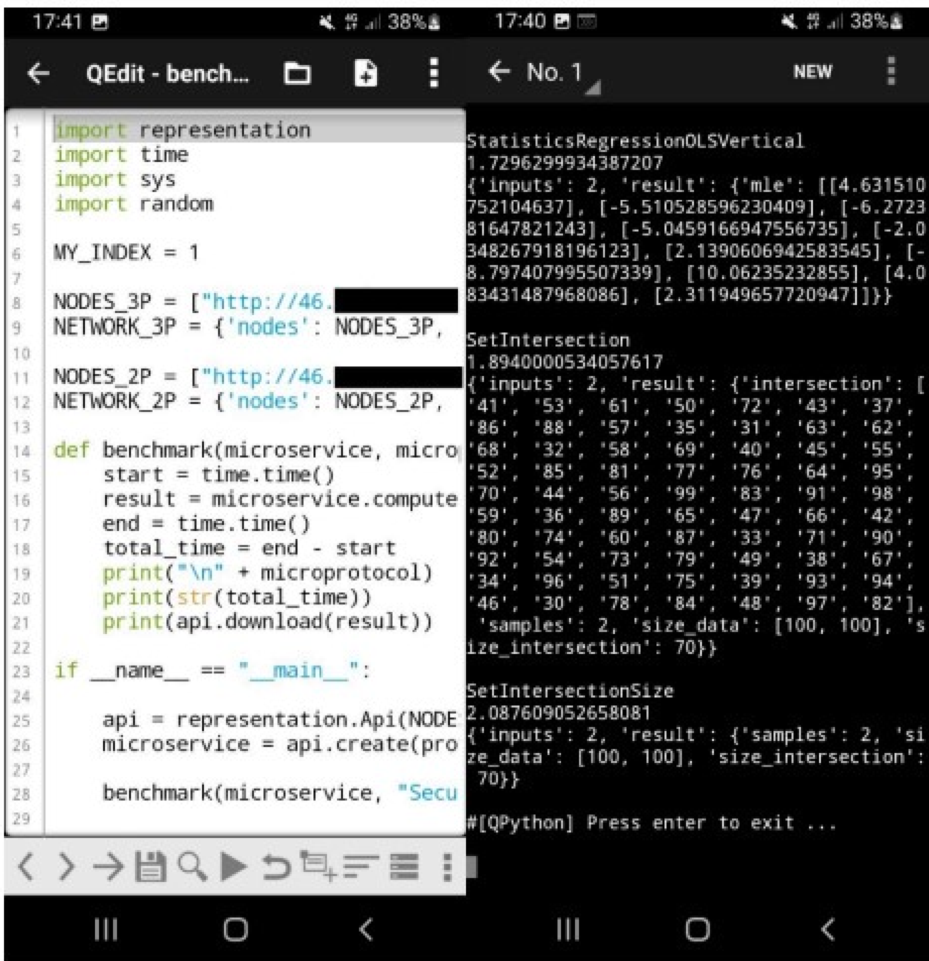


Figure 8. Creating and executing a client script on an Android smartphone

Alternative Computing Frameworks

In the preceding sections, we employed "Simon" (SImple Multiparty computatiON) as an illustration of secure multiparty computation within a peer-to-peer network. Simon is included as a package in the primary repository under a free license. Its primary purpose is to serve as a rapid educational example without necessitating more intricate third-party backends. Nevertheless, there has been exploration involving other approaches and configurations

Homomorphic Encryption with Amazon Web Services

In an initial implementation of Federated Secure Computing in 2020, a comparison was drawn between the performance of earlier efforts [23–25] and a novel architecture based on dedicated microservices. Concerns at that time included the level of effort required for implementation, as well as the achievable precision and computing time.

During this phase, a prototype of Federated Secure Computing was deployed in a public cloud environment (Amazon Web Services, AWS, Seattle, WA, USA) using "serverless" microservices (AWS Lambda). This implementation incorporated "Sophie," designed for "simple homomorphic encryption," essentially forming a trusted computing framework with a degree of privacy protection through local aggregation and homomorphic obfuscation.

Leveraging a public cloud provider, implementing serverless microservices, employing the public cellular network, utilizing consumer-grade hardware, and adopting the Python language constituted notable conceptual achievements. The encouraging performance results provided impetus for the ongoing development of Federated Secure Computing, as detailed in Table 13.

Table 13. Comparison between secure multiparty computation and an early version of Federated Secure Computing.

	Earlier Work [12–18,20–25]	Federated Secure Computing
data	• real patient data	• pseudodata
routing	• Munich to Berlin (500 km) • LMU to Charité • fast internet backbone	• Munich to Frankfurt (300 km) • LMU to AWS eu-central-1 • public 4G mobile network
protocol	• secure multiparty computation	• (weak) homomorphic encryption
software	• FRESCO, SPDZ	• fdrtd 0.3.1 (build 2020-03-02)
hardware	• Intel Xeon Silver 4112 CPU • 8 cores, 2.6 GHz, 8 MB cache • 128 GB RAM • 1 Gb/s ethernet networking	• Samsung Galaxy Xcover 4 • Exynos 7570 CPU, 4 × 1.40 GHz • 2 GB RAM • ~100 Mb/s LTE networking
implementation effort	• several months of preparation • two systems administrators • one crypto expert programmer	• around 15 min • 9 lines of Python code
achieved precision	• 1.8% to 11.7% relative numerical error	• exact to floating point machine precision
runtime	• 1229 s	• 4.5 s

Differential Privacy with DataSHIELD as Backend

Considering its utilization in the German Medical Informatics Initiative (MII), the trusted computing framework DataSHIELD emerges as a potentially valuable backend choice. A student, as part of a third-party-funded project, implemented an interface to DataSHIELD comprising microservices. The demonstration showcased the functionality, and calculations were executed successfully. The performance aligned with the raw DataSHIELD performance, exhibiting no discernible overhead from the middleware. A revised version of this interface may be incorporated into the Federated Secure Computing repository in the future.

Secure Multiparty Computation with Sharemind

Sharemind stands out as a robust, industry-grade solution for secure computing. It employs secure multiparty computation, ensuring end-to-end data protection and accountability. Sharemind is

presently under evaluation as a comprehensive backend solution, with the development of an interface underway.

DISCUSSION

Project Background and Ownership

In 2019, we conducted the world's inaugural secure multiparty computation (SMPC) calculation with real patient data, connecting the university hospitals of LMU Munich and Charité Berlin [23–25]. A specialized security researcher from TUM had to develop custom cryptography based on the FRESCO/SPDZ framework [26–28]. System administrators were required to manage dedicated hardware servers, and the entire process, from preparation to execution, spanned several months. Our experience affirmed the viability of privacy-preserving computation but underscored its complexity.

In 2020, interviews with companies and government agencies echoed our findings. The prospect of enabling secure analysis without data sharing or reliance on a trusted third party was appealing. However, there was substantial hesitancy in adopting a technology that developers and data security officers poorly understood, with exotic tech stacks and skills gaps often proving insurmountable. Simultaneously, a market survey revealed the availability of robust and powerful privacy-preserving computation (PPC) frameworks, both open source and proprietary. Most potential use cases explored required basic algorithms well-documented in the literature. Thus, it became evident that the necessary technology already existed. What was lacking was a simple middleware to enhance the DevSecOps experience.

The initial endeavor, "Multiparty Computation as a Service," sought to decouple the cryptography layer from business logic, shifting the complex and computing-intensive workload to the cloud. Successful testing in May 2020 at a private healthcare company involved installation at two remote locations within 15 minutes and the execution of a distributed analysis within seconds.

In 2021, bytes for life GmbH, a cloud computing startup based in Munich, conceived "Federated Secure Computing," the architecture detailed in this paper, and proceeded to develop a reference implementation. The technology was contributed to LMU University Hospital for "Wirkung hoch 100," a national competition by Stifterverband (the German donors' association) with the explicit goal

of enhancing the German education, science, and innovation ecosystem. Over the course of a year, the team iteratively refined their solution for systemic impact. Throughout this process, "Federated Secure Computing" was released as free and open source. In November 2021, Stifterverband recognized the innovation by awarding its Innovation Prize to Federated Secure Computing and providing three years of funding to further advance the technology [29–31].

In 2022, bytes for life GmbH transferred the complete intellectual property and all assets to Ludwig-Maximilians-Universität München (LMU Munich). As an esteemed university of excellence, LMU Munich is well-positioned to host the open source project for the long term and champion its scientific and economic development.

Project Status

At the time of this paper's writing, most design objectives have been realized. Concerning architecture, various topologies accommodating any number of data, compute, and research nodes are feasible, maintaining a strict client–server separation of cryptography and business logic concerns.

In terms of implementation, the server-side reference implementation is 100% Python without relying on exotic dependencies. The client side is non-opinionated, and stubs for the OpenAPI interface are accessible for over a dozen programming languages. The software has undergone testing in propaedeutic settings and proved usable by non-expert programmers.

Regarding functionality, there exists a working implementation of a propaedeutic Simple Multiparty Computation (Simon) protocol offering frequently used algorithms as a collection of microservices. Currently, microservices are available for secure sum and secure matrix multiplication, private set intersection, private set intersection size, univariate and bivariate statistics, frequency histograms and contingency tables, and ordinary least squares regression.

In the realm of DevOps, the software has been successfully installed and run within minutes on hardware ranging from dedicated servers to heterogeneous clients to IoT devices. Crafting a simple, secure computation requires fewer than a dozen lines of user code.

Concerning IoT, both the client-side and server-side software are lightweight enough to operate on Raspberry Zero devices or last-generation (G3) smartphones. Testing has been conducted on devices as early as the Samsung Galaxy XCover 2 model (GT-S7710) from 2013.

In terms of Free and Open Source Software (FOSS), the software is distributed under a permissive MIT license. The intellectual property is owned by LMU Munich, a reputable university and non-profit institution recognized by the State of Bavaria under German and Bavarian law.

For accessibility, there is a dedicated project website, a GitHub organization with a structured dashboard and repositories, a variety of easily installable PyPI packages, and this comprehensive whitepaper.

CONCLUSION

In summary, "Federated Secure Computing" serves as both an initiative and a middleware with the goal of democratizing Privacy-Preserving Computation (PPC) accessibility, targeting small and medium enterprises, startups, research and teaching institutions, and government agencies. This technology is provided as free and open source software, supported by Ludwig-Maximilians-Universität München and funded by Stifterverband.

This whitepaper is intended to facilitate engagement with the Federated Secure Computing ecosystem, and we encourage individuals from both the public and private sectors to explore its potential applications. Contributions to the codebase are warmly welcomed, fostering collaborative growth and development.

BIBLIOGRAPHICAL REFERENCES

Deloitte. The Analytics Advantage; Deloitte: London, UK, 2013.

AIG. The Data Sharing Economy: Quantifying Tradeoffs That Power New Business Models; AIG: New York, NY, USA, 2016.

European Commission. Study on Data Sharing between Companies in Europe. 2018. Available online: https://op.europa.eu/s/y2R4 (accessed on 23 August 2023).

TrustArc. TRUSTe/National Cyber Security Alliance U.S. Consumer Privacy Index; TrustArc: San Francisco, CA, USA, 2016.

Yao, A.C. Protocols for secure computations. In Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, Chicago, IL, USA, 3–5 November 1982; pp. 160–164.

Damgard, I.; Pastro, V.; Smart, N.; Zakarias, S. Multiparty Computation from Somewhat Homomorphic Encryption. In Proceedings of the 32nd Annual International Cryptology Conference (CRYPTO), University of California Santa Barbara, Santa

Barbara, CA, USA, 19–23 August 2012; pp. 643–662.

Gentry, C. Fully Homomorphic Encryption Using Ideal Lattices. In Proceedings of the 41st Annual ACM Symposium on Theory of Computing, Bethesda, MD, USA, 31 May–2 June 2009; pp. 169–178.

Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In Advances in Cryptology—Eurocrypt'99; Stern, J., Ed.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 1999; Volume 1592, pp. 223–238.

Dwork, C. Differential privacy. In Automata, Languages and Programming, Pt 2; Bugliesi, M., Prennel, B., Sassone, V., Wegener, I., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2006; Volume 4052, pp. 1–12.

Dwork, C.; Roth, A. The Algorithmic Foundations of Differential Privacy. Found. Trends Theor. Comput. Sci. 2013, 9, 211–406.

[CrossRef]

Craddock, M.; Archer, D.W.; Bogdanov, D.; Gascon, A.; de Balle Pigem, B.; Laine, K.; Trask, A.; Raykova, M.; Jug, M.; McLellan, R.; et al. UN Handbook on Privacy-Preserving Computation Techniques. 2019. Available online:

https://unstats.un.org/bigdata/task-teams/privacy/UN%20Handbook%20for%20Privacy-Preserving%20Techniques.pdf (accessed on 23 August 2023).

Kolesnikov, V.; Schneider, T. Improved garbled circuit: Free XOR gates and applications. In Proceedings of the 35th International

Colloquium on Automata, Languages and Programming, Reykjavik, Iceland, 7–11 July 2008; pp. 486–498.

Shpilka, A.; Yehudayoff, A. Arithmetic Circuits: A Survey of Recent Results and Open Questions. Found. Trends Theor. Comput.

Sci. 2009, 5, 207–388. [CrossRef]

Bernstein, D.J. Curve25519: New Diffie-Hellman speed records. In Public Key Cryptography—Pkc 2006, Proceedings; Yung, M., Dodis, Y., Kiayias, A., Malkin, T., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2006; Volume 3958,

pp. 207–228.

Diffie, W.; Hellman, M.E. New directions in cryptography. IEEE Trans. Inf. Theory 1976, 22, 644–654. [CrossRef]

Rabin, M.O. How to Exchange Secrets with Oblivious Transfer; Aiken Computation Laboratory, Harvard University: Cambridge, MA, USA, 1981.

Bogdanov, D.; Laur, S.; Willemson, J. Sharemind: A Framework for Fast Privacy-Preserving Computations. In Proceedings of the 13th European Symposium on Research in Computer Security, Malaga, Spain, 6–8 October 2008; pp. 192–206.

Wirth, F.N.; Kussel, T.; Muller, A.; Hamacher, K.; Prasser, F. EasySMPC: A simple but powerful no-code tool for practical secure multiparty computation. BMC Bioinform. 2022, 23, 531. [CrossRef]

Gay, W. Raspberry Pi Hardware Reference; Apress: Berkeley, CA, USA, 2014.

Gaye, A.; Marcon, Y.; Isaeva, J.; LaFlamme, P.; Turner, A.; Jones, E.M.; Minion, J.; Boyd, A.W.; Newby, C.J.; Nuotio, M.L.; et al.

DataSHIELD: Taking the analysis to the data, not the data to the analysis. Int. J. Epidemiol. 2014, 43, 1929–1944. [CrossRef]

[PubMed]

Wolfson, M.; Wallace, S.E.; Masca, N.; Rowe, G.; Sheehan, N.A.; Ferretti, V.; LaFlamme, P.; Tobin, M.D.; Macleod, J.; Little, J.; 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, 1372–1382. [CrossRef] [PubMed]

The Linux Foundation. New Collaborative Project to Extend Swagger Specification for Building Connected Applications and Services. 2015.

Available online: https://www.linuxfoundation.org/press/press-release/new-collaborative-project-to-extendswagger-specification-for-building-connected-applications-and-services (accessed on 23 August 2023).

Krüger-Brand, H.E. Innovatives IT-Verfahren soll sensible Daten in der Krebsforschung schützen. Ärzteblatt. 2019. Available

online: https://www.aerzteblatt.de/nachrichten/103090/Innovatives-IT-Verfahren-soll-sensible-Daten-in-der-Krebsforschungschuetzen (accessed on 23 August 2023).

Ballhausen, H.; von Maltitz, M.; Niyazi, M.; Kaul, D.; Belka, C.; Carle, G. Secure Multiparty Computation in Clinical Research and

Digital Health. In Proceedings of the E-Science-Tage 2019, Heidelberg, Germany, 27–29 March 2019.

von Maltitz, M.; Ballhausen, H.; Kaul, D.; Fleischmann, D.F.; Niyazi, M.; Belka, C.; Carle, G. A Privacy-Preserving Log-Rank

Test for the Kaplan-Meier Estimator With Secure Multiparty Computation: Algorithm Development and Validation. JMIR Med.

Inform. 2021, 9, e22158. [CrossRef] [PubMed]

Keller, M. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In Proceedings of the ACM SIGSAC Conference on

Computer and Communications Security (ACM CCS), Virtual Event, 9–13 November 2020; pp. 1575–1590.

Keller, M.; Pastro, V.; Rotaru, D. Overdrive: Making SPDZ Great Again. In Proceedings of the 37th Annual International

Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), Tel Aviv, Israel, 29 April–3 May 2018;
pp. 158–189.

Keller, M.; Scholl, P. Efficient, Oblivious Data Structures for MPC. In Proceedings of the 20th Annual International Conference
on the Theory and Application of Cryptology and Information Security (Asiacrypt), Kaoshiung, Taiwan, 7–11 December 2014;
pp. 506–525.

Niebuhr, C. Daten tauschen und schützen—Das muss kein Widerspruch sein. MERTON. 2021. Available online: https://mertonmagazin.de/daten-tauschen-und-schuetzen-das-muss-kein-widerspruch-sein (accessed on 23 August 2023).

LMU-Forschende mit Ideen zu Innovation und Bildung Erfolgreich. Available online: https://www.lmu.de/de/newsroom/newsuebersicht/news/lmu-forschende-mit-ideen-zu-innovation-und-bildung-erfolgreich.html (accessed on 22 September 2021).

Federated Secure Computing. Available online: https://www.stifterverband.org/wirkunghoch100/3projekte/computing
(accessed on 16 August 2023).