# Secure Agents*

Nakul Khambhati
nakul@cs.ucla.edu

Joonwon Lee
jlee2023@g.ucla.edu

Gary Song
gary.song@g.ucla.edu

Rafail Ostrovsky
rafail@cs.ucla.edu

Sam Kumar
samkumar@cs.ucla.edu

*University of California, Los Angeles*

**Abstract**

Organizations increasingly need to pool their sensitive data for collaborative computation while keeping their own data private from each other. One approach is to use a family of cryptographic protocols called Secure Multi-Party Computation (MPC). Another option is to use a set of cloud services called clean rooms. Unfortunately, neither approach is satisfactory. MPC is orders of magnitude more resource-intensive than regular computation, making it impractical for workloads like data analytics and AI. Clean rooms do not give users the flexibility to perform arbitrary computations.

We propose and develop an approach and system called a *secure agent* and utilize it to create a virtual clean room, Flexroom, that is both performant and flexible. Secure agents enable parties to create a *phantom identity* that they can collectively control, using maliciously secure MPC, which issues API calls to external services with parameters that remain secret from all participating parties. Importantly, in Flexroom, the secure agent uses MPC not to perform the computation itself, but instead merely to *orchestrate* the computation in the cloud, acting as a distinct trusted entity jointly governed by all parties. As a result, Flexroom enables collaborative computation with unfettered flexibility, including the ability to use convenient cloud services. By design, the collaborative computation runs at plaintext speeds, so the overhead of Flexroom will be amortized over a long computation.

*The first three authors are students with equal contribution; the last two authors are faculty jointly guiding the project.

# 1 Introduction

Organizations increasingly wish to collaborate by pooling their data and running workloads, such as AI training and data analytics, on their combined dataset. Furthermore, they often need to do this while keeping their data private from the other organizations in the collaboration. For example, multiple banks need to pool their data sets to detect money laundering [8, 37] (a type of fraud that involves transferring money between institutions), but cannot directly reveal the data of their customers to each other. Similarly, evaluating treatments for rare diseases can require collaboration between multiple government agencies to obtain a large sample size [19], but patient data cannot be freely shared due to concerns about privacy.

An approach to supporting such collaborative workloads is to utilize cryptographic protocols for computing on encrypted data, including the use of a family of protocols known as *Secure Multi-Party Computation (MPC)*. Many systems have been proposed for this in academia [3,17,20,21,29,34,37], and MPC-based techniques are beginning to see real-world adoption with systems like Meta's Private Lift [24], Google's Private Join and Compute [31], and others [22].

Unfortunately, MPC remains orders of magnitude slower than regular (plaintext) computation [16, 29], and is not practical for certain important workloads. For collaborative data analytics, MPC scales only to moderately-sized datasets, even with system-level techniques designed to minimize the size of the MPC computation [3, 29]. For AI training and inference using MPC, the overheads of MPC place a practical limit on the size and complexity of models that can be used. For example, it takes $\approx 1.5$ *days* to train a VGG16 model on the CIFAR-10 dataset using MPC [34]; this is two-to-three orders of magnitude slower than training in plaintext, which takes only a few minutes [30]. Furthermore, these results are in a LAN setting with plentiful network bandwidth and low network latency; the practical overheads of MPC are even higher when parties must communicate over wide-area networks or the public Internet.

To address this issue, cloud providers recently (in $\approx$ 2023) began providing *clean room* products [2, 12] that allow multiple cloud tenants to engage in a collaborative computation. These products provide weaker security than MPC-based solutions because the cloud provider acts as a trusted party that performs the computation on the tenants' behalf. However, unlike MPC, clean rooms allow collaborative computation at fast, plaintext speeds and with the convenience of cloud services. In principle, clean rooms would enable any computation that a single user can run in the cloud to also be run collaboratively with the same performance and convenience.

Unfortunately, clean rooms do not achieve this potential in practice, because they lack *flexibility*—they only allow users to perform computations that are specifically clean-room-enabled. For example, AWS Clean Rooms supports running queries over multiple organizations' combined dataset, with added differential privacy, but not training an arbitrary ML model on the combined dataset. In contrast, a single non-collaborative user can use general-purpose platforms like VMs, and can even use the vast library of SaaS products of their chosen cloud provider. Clean rooms could, in principle, be extended to include general-purpose platforms like VMs, but, in practice, they have significantly lagged behind the vast array of cloud services available to a non-collaborative user.

In this context, we ask: **How can we enable collaborative computation that is both *performant* and *flexible*?** To answer this, we develop a new system design element called a *secure agent*, and use it to design a new clean room, called Flexroom. Flexroom enables *any* cloud service in Amazon Web Services (AWS) to be used in a collaborative setting.

### 1.1 Key Design Element: Secure Agents

**Strawman.** To explain Flexroom, we begin with a simple strawman design. This strawman design relies on a party distinct from the cloud that we call the Clean Room Service Provider (CRSP). Users wishing to collaborate send their data and a description of their computation (e.g., a data analytics or AI training task) to the CRSP. The CRSP has an account with a cloud provider and maintains the credentials (e.g., username/password, API token, etc.) for that account. The CRSP accesses cloud services using that account to perform the users' computation. Unfortunately, this starting point provides weaker security than a traditional clean room: Users must trust not only the cloud provider, but also the CRSP. An attacker who has compromised the CRSP can access users' data. This is particularly problematic if the CRSP is less established or reputable than the large cloud providers with clean room services today (AWS, GCP).

**Our Design.** Flexroom solves this by *using MPC to perform the functionality of a CRSP* in the above strawman design. Specifically, whereas prior MPC-based collaboration systems use MPC directly to perform computation on the combined dataset, Flexroom uses MPC to decide which API calls to issue to the cloud provider. To highlight this distinction, we refer to this use of MPC as a *secure agent*. Secure agents allow users to jointly issue API calls to an external service, where predefined, agreed-upon functions determine how the API calls are constructed based on the users' inputs and which parts of the response, if any, are revealed to the users. For Flexroom, the secure agent has a cloud account, within which the collaborative computation takes place (just like the CSRP's cloud account in the strawman).

In Flexroom, the performance costs of MPC affect only the *orchestration* of the collaborative computation and is independent of the computation's running time; the computation itself happens in plaintext in the cloud. Thus, for large computations where the computation itself dominates the runtime, Flexroom has only a modest performance cost. Yet, because any cloud service can be accessed via API calls, Flexroom allows cloud services available to an ordinary cloud tenant to be used in a collaborative setting.

### 1.2 Achieving "Security" for Secure Agents

In the context of Flexroom, our secure agents must only issue an API call to the external service (i.e., the cloud provider) if there is *unanimous agreement* among the users to issue that API call. Unanimous agreement allows a user to share their data with the secure agent's cloud account, and then the secure agent will only issue API calls that use their data appropriately, as part of an agreed-upon collaborative computation. In particular, the other users cannot work together to issue API calls that disclose that user's data. To achieve this security property, we design secure agents using **three techniques**, which we describe below.

Our first technique is that **we create secure agents using *maliciously-secure, dishonest-majority MPC.*** Because the API calls are constructed within MPC, these properties guarantee that *all* users agree on exactly which procedure is used to construct each API call. Thus, the secure agent will not issue an API call to the external service (i.e., the cloud provider) unless *all* users agree on how to construct that API call.

This first technique is not sufficient on its own because, in our design so far, the secure agent lacks *complete mediation*—a malicious user can simply bypass the secure agent by issuing API calls on its own, without going through the secure agent. This leads to our second technique: **we use reactive MPC techniques to give secure agents access to *private mutable state*, and**

**store cloud credentials within that private mutable state**. Importantly, the secure agent's private mutable state is secret-shared [26] among the users, so that it is accessible within MPC but not by any subset of the users. Because the users do not know the credentials, they cannot construct valid API calls to the cloud provider on their own; they must rely on the secure agent (i.e., MPC) to compute such valid API calls.

A third challenge is that the external service (i.e., the cloud provider) expects API calls to be issued as a single user would—via a single network connection. Thus, one of the users must use their physical computer to establish a network connection with the cloud endpoint to send the request output by the secure agent, and receive the response. That user could snoop on the requests to learn the secure agent's cloud credentials (e.g., username and password) and then construct its own malicious requests, to hijack the cloud tenant and reveal the other users' data. We address this with our third technique: **we *run a TLS client inside of MPC* for the secure agent. Specifically, the secure agent outputs each API call not as plaintext, but as a TLS-secured byte stream to send to the server.** With this design, the user who establishes the connection is no more powerful than a man-in-the-middle attacker, against which TLS guarantees confidentiality and integrity, thereby thwarting the attack that we described above.

### 1.3 Interacting with the Cloud API Endpoint

Using secure agents with a real cloud provider brings additional practical challenges that Flexroom must address.

The first challenge is that cloud accounts are tied to *real world information* and *non-cryptographic societal protocols* that make it difficult to ensure complete mediation for the secure agent. The cloud provider expects an account to be associated with real-world user information such as a name, email address, billing information, etc. Simply using the information of one of the users is insecure, because it would allow that user to leverage their knowledge of that information to hijack the account. For example, that user could contact the cloud provider's customer service to reset the account's password, sharing their information and proving they can access their email account to convince the representative that the cloud account is theirs, thereby gaining total access to the virtual cloud tenant and the other users' data. To address this, our insight is to have the secure agent create and control not only a cloud account, but also a *phantom identity* encompassing all aspects of user information associated with that account. Specifically, the secure agent would choose a random name and issue API calls to create a fresh email address with a random password. This information would be stored within the secure agent's state, so that it can only be accessed within MPC to produce API calls and not by any individual user. This can be achieved by secret-sharing [26] the information among the users. We also propose a solution (which we do not implement) for hiding billing information from the users. Billing information is tricky; while the users can create a fresh email account for a phantom identity, they cannot legally create an anonymous bank account, credit card account, or Social Security Number due to "know your customer" regulations. Our solution is to have the secure agent issue API calls to purchase a pre-paid debit card (accepted by AWS) that is not associated with any of the users. Phantom identities enable secure agents, in principle, to access a variety of online services, including those that require user accounts and billing.

The second challenge is that cloud APIs evolve over time and are extended frequently as new cloud services become available. Keeping up with the latest advances requires significant development effort, to implement new API calls in MPC. Our insight is that the *completeness* of cloud computing can simplify this. Specifically, once the secure agent issues an API call to spawn a cloud

VM (e.g., in EC2), it can place credentials to access the cloud account on the allocated VM and program the VM to use other cloud services under that account. The code running on the VM does not need to issue API calls from MPC; instead, it can simply use off-the-shelf cloud libraries to issue API calls normally (e.g., using AWS' `boto3` library). This allows a relatively simple implementation for Flexroom; even though a cloud provider may offer a very rich API, only a very small subset of that API—enough to spawn a VM and execute code on it—must be supported within MPC.

### 1.4 Summary of Evaluation

We implemented Flexroom for $n = 2$ users running a computation using Amazon Web Services (AWS). We evaluated it in a setting where the two users using Flexroom are located in different but nearby regions of Microsoft Azure. The process of signing into the account, to obtain an API token in the secure agent's private storage, takes $\approx 45$ minutes (excluding account creation); this is a one-time process, and the resulting session can be reused for multiple computations. The process of spawning a VM in EC2 and starting a computation takes $\approx 7$ minutes. However, the collaborative computation itself runs in plaintext in the cloud without additional slowdown, so this cost of starting a job is amortized over the course of a long computation.

## 2 Background

### 2.1 Transport Layer Security (TLS)

We first recap the TLS protocol which is run between a client and a server to allow transmission of data in a way that preserves privacy and integrity. Flexroom supports TLS 1.3, the leading standard for communication over the internet. TLS, by itself, does not allow for collaborative computing or computing on encrypted data. Flexroom runs the client side of the protocol within an MPC engine run jointly by two parties while the server side protocol is unmodified.

TLS consists of two subprotocols: the handshake, which establishes shared symmetric keys and negotiates cryptographic algorithms, and the record layer, which protects all subsequent communication using these keys. The record layer applies authenticated encryption with additional data (AEAD) to ensure both confidentiality and integrity of transmitted data. Nonces used in AEAD are deterministically derived from an initialization vector, meaning only the key itself must remain secret.

#### 2.1.1 TLS 1.3 handshake

The handshake proceeds in two stages: key exchange and authentication. In the key exchange, the client initiates communication with a ClientHello containing a nonce, supported algorithms, and a Diffie–Hellman public key. The server responds with its own nonce, selected algorithms, and a Diffie–Hellman key. Both parties then derive shared secrets from these values and the transcript (the concatenation of exchanged messages), which yield temporary "handshake keys" for secure communication during setup. In the authentication phase, the server exchanges its certificate and digitally signs the transcript to prove ownership of its keys. Any failure in this check terminates the connection.

### 2.1.2 TLS 1.3 record layer

Once authentication completes, the handshake messages are fed into the key derivation function again to produce fresh application keys and initialization vectors. These replace the handshake keys and are used exclusively by the record layer to secure client-to-server and server-to-client traffic. The design ensures that even small transcript changes result in entirely different keys, and the outputs of the handshake remain computationally independent. This offers protection even against man-in-the-middle attackers who can observe and tamper with transmitted data packets. Flexroom supports AES in CBC-HMAC and GCM modes. The reader is referred to [23] for more details about the TLS protocol.

## 2.2 Secure Multi-Party Computation (MPC)

Secure multi-party computation allows a group of mutually distrusting parties to jointly evaluate a function on private inputs, ensuring that no party learns anything but the output of the function. MPC comes in various flavors and this work provides security with abort against a dishonest majority of malicious parties, which is the strongest notion of security for two party computation. These protocols can tolerate adversaries who arbitrarily deviate from the protocol in order to break the inputs' privacy and/or the outputs' correctness.

MPC's security is defined in the simulation-based paradigm [5] which guarantees that messages exchanged by parties during a protocol reveal nothing about their private inputs except what can be deduced from the output of the computation. The MPC functionality essentially replaces a trusted party who obtains private inputs from multiple parties, computes on data and broadcasts only the output. It is possible to generalize the behavior of this MPC functionality so that it receives private inputs from parties over time and performs intermediate computation, keeping its own private internal state between rounds. Such a functionality is called *reactive* [10] and various MPC protocols can instantiate this without any overhead.

To engage in an MPC protocol, parties need to agree on a computational domain for the function they wish to compute. Some examples are large finite fields, [15], binary circuits [35], arithmetic modulo a power of two [6] and elliptic curve groups [7]. These are cost-optimized for different applications such as integer computation like addition, non-linear computation like comparisons and protocols over elliptic curves such as Diffie Helman key exchange. The TLS protocol features both linear functionality over an elliptic curve and non-linear functionality, best represented using boolean circuits. It is therefore desirable to convert between an arithmetic computation domain and binary circuits. Double-authenticated bit (daBit) [9, 25] is a general technique that authenticates the same bit in two domains, e.g., $\mathbb{Z}_p$ and $\mathbb{F}_2$. This also inspired double-authenticated point (daPoint) [1], an efficient protocol to convert between authenticated group shares of a point and field shares of its affine coordinates. We build on prior work to implement these conversion techniques which greatly benefit the performance of our system.

In this work, we run TLS 1.3 client within an MPC engine which has been proposed in prior works [1, 28, 36] but none of these works provide an open-source implementation that can be used directly for Flexroom. We explain our approach for implementing TLS-in-MPC in Section 4.

## 2.3 Clean Rooms

Cloud-based clean rooms enable multiple parties to collaborate at near-plaintext speed by relying on a trusted, neutral platform, the cloud provider, to orchestrate data analysis and manage data. In

this cloud environment, each party's sensitive data is either uploaded or linked within a protected cloud infrastructure, where computations are executed under strict access control enforced by the service. Consequently, the security of the entire collaborative process and the shared data is heavily dependent on the trustworthiness and policies of the cloud operator.

In contrast, heavy computation tasks have also been executed within an MPC setting. However, the cryptographic overhead inherent to MPC renders intensive workloads, such as AI training, practically infeasible to carry out in an MPC environment.

# 3 System Overview

## 3.1 System Architecture and Secure Agents

Our approach supports an arbitrary number $n$ of participating users. Their MPC computation has access to persistent state in the form of a small database secret-shared among the $n$ users (as is standard in reactive MPC protocols). Specifically, the $n$ users can use MPC for any functionality that they both agree to, and that functionality can read and write the contents of this database. Importantly, no subset of the $n$ users can directly access the contents of the database—the database is only accessible via an MPC computation involving all $n$ users.

Our design generalizes to any $n > 1$, but the cost of orchestrating the computation scales with the number of users. [14, 33]. We implemented the protocol for the case of $n = 2$. For simplicity, the rest of our explanation in this paper is specific to the $n = 2$ case, and we refer to the two users as Alice and Bob.

In this setup, Alice and Bob use MPC to run a TLS client. Alice and Bob can access HTTPS-enabled web servers from inside MPC by having one party—Alice in this example—transfer encrypted data between the web server and the MPC computation containing the TLS client. When the TLS client wishes to send a message to the web server as part of the TLS protocol, it outputs that message from MPC and saves a continuation (i.e., a description of the TLS protocol state) to memory; Alice forwards the message to the web server. When the web server responds with a message in the TLS protocol, Alice provides it as input to MPC, which uses the continuation stored in memory to process (e.g., decrypt) it.

This system architecture enables Alice and Bob to use MPC to issue API calls to external services, using data in their secret database to construct those API calls—a design element we refer to as a *secure agent*. The key idea is that the secret database contains credentials used to make API calls (e.g., email address, username, password) that are not known to either Alice or Bob, so Alice and Bob can *only* issue the desired API calls via the secure agent. Yet, because the secure agent is controlled using MPC, Alice and Bob must *unanimously agree* on any API call. Because the API call contains credentials that cannot be disclosed to either Alice or Bob, agreement is achieved by having Alice and Bob agree on a *functionality* that generates the API call based on the secure agent's private state.

## 3.2 Flexroom

Flexroom is a clean room built using secure agents. First, Alice and Bob run **FlexroomInit** to initialize their secret-shared database, which we denote as `state`. They can run three operations that operate on their `state`, and possibly modify their state. The first operation, `state`.**CreateAccount**, creates a phantom identity and an account with the cloud provider. The credentials are not known

directly to either Alice or Bob, and are stored in the secret-shared database. The second operation, state.**CreateSession**, signs in to the cloud provider using the secret credentials, and stores the session tokens (e.g., cookies) in the secret-shared database. The third operation, state.**RunScript**, uses MPC to execute a Python script, which has access to the session tokens and cloud credentials and can issue API calls to allocate and use cloud resources. Importantly, both Alice and Bob must agree on the exact operation to perform and the arguments to that operation (e.g., the script to be run for **RunScript**) in order for that operation to be performed. Thus, Flexroom allows Alice and Bob to jointly control a cloud tenant that can allocate cloud resources and perform any action that both Alice and Bob agree to.

Flexroom offers a middle ground between conventional clean room services and pure MPC environments. It mitigates the vulnerabilities associated with complete reliance on the cloud operator by employing a hybrid architecture that leverages MPC to collaboratively generate and verify API calls.

### 3.3 Threat Model and Security Guarantees

Flexroom's security hinges on the security of TLS 1.3 and the MPC engine used to instantiate the TLS 1.3 client. Flexroom is secure against a malicious adversary who corrupts up to $n-1$ out of $n$ participating users. Specifically, it ensures that:

- No party can issue any API call to the cloud provider on behalf of the secure agent's cloud tenant unless all parties agree on the functionality to generate that API call.
- When successfully issuing API calls with unanimous agreement, the adversary learns only the length of the API call and its response. Further, the adversary can issue an abort at any point in the protocol.

Flexroom guarantees this in a *malicious* model—it assumes that the adversary can control corrupted parties' behavior, causing them to deviate arbitrary from the specified protocol. For example, if the adversary controls the party that establishes the TLS endpoint with the server, then it may send maliciously crafted messages to honest parties or to external web servers that Flexroom is communicating with, and drop or delay messages that the endpoint would normally send.

The security guarantees follow from the fact that we use *maliciously-secure* MPC to run the secure agent. This means that if any subset of parties is malicious and tries to interfere with the MPC protocol, the worst they can do is to deny service (i.e., cause the computation to abort)—they cannot learn the contents of the secret database or any intermediate results of executing the functionality run using MPC. More formally, any information that an adversary learns by acting maliciously can be simulated using only the output of the MPC engine, which in this case consists of messages communicated in the TLS protocol. Next, we appeal to the fact that the secure agent communicates with external services exclusively via TLS, which is designed to be resilient to man-in-the-middle attacks. Thus, even though an adversary who controls Alice can drop, inject, or modify messages exchanged between the secure agent and an external web service, these messages are protected by TLS, which ensures that the worst that the adversary can do is deny service.

Flexroom does not protect against denial-of-service attacks. The adversary can destroy the secret state or prevent the execution of future Flexroom-supported operations (**CreateAccount**, **CreateSession**, **RunScript**). For certain uses of Flexroom, this could lead to resource waste—for example, if Alice and Bob use **RunScript** to start a cloud VM, an adversary controlling either party could prevent future executions of **RunScript** that would normally terminate the VM. The

parties can write their scripts defensively (e.g., terminating all resources after a timeout) to avoid resource waste that would result from such attacks.

Flexroom also does not protect against side channels. For example, the adversary may be able to learn some information about the contents of `state` by observing the timing of messages sent from the web service that Flexroom is communicating with, or the sizes of messages that they exchange. Flexroom also does not protect against an adversary who has compromised the cloud provider, or who has compromised any web service with which Flexroom communicates (e.g., an email server used in **CreateAccount**).

# 4 System Design

## 4.1 TLS Client in MPC

Our system builds on prior work, Oblivious TLS [1], that provides a theoretical foundation for running TLS in MPC. Another prior work, MPCAuth [28], provides measurements of an implementation of TLS in MPC, but the implementation is not open-source or publicly available and their maliciously-secure share conversion protocols are not fully specified. We also requested the authors of MPCAuth for their implementation, but they did not provide it to us.

We designed and implemented a protocol for running a TLS client in MPC based on Oblivious TLS [1]. To implement a TLS client within the MPC framework, we must efficiently handle each of the several components of the TLS protocol using MPC. As explained briefly in Section 2, the TLS handshake establishes secure parameters between communicating parties. Running this functionality inside MPC presents some efficiency challenges. Our implementation divides the protocol across two phases based on the computation domain that is more favorable.

### 4.1.1 TLS Handshake in MPC

We use ECDH for the TLS handshake as described in Section 2. Doing this inside MPC requires an exponentiation between a known public key and a secret exponent, where the output must remain secret. The shared key is an EC point, which must be represented in a suitable manner in the MPC engine. We use the SPDZ protocol extended to handle authenticated shares over elliptic curve groups [27] for this step. We refer readers to [1] for a detailed description of the MPC Diffie Helman key exchange protocol. Next, we use Oblivious TLS's daPoint protocol [1] to efficiently convert between authenticated secret shares of the EC point over an EC group to authenticated secret shares of its affine coordinates over a finite field. We then use the extended daBit protocol [9] to efficiently convert between finite field shares and garbled circuit shares of the point. This allows parties to feed the inputs into a garbled circuit [35] protocol which efficiently performs computation suited for binary domains. The remainder of the TLS handshake, including key derivation and AES-GCM can be done efficiently using boolean operations. We reveal the server's certificate in the clear to both users who verify its integrity outside MPC. Prior works [28, 36] observe that this is safe to do and does not violate the security guarantees of TLS.

### 4.1.2 TLS Record Layer in MPC

The TLS record layer, responsible for encrypting and authenticating application data, similarly operates in the boolean domain using garbled circuits. Our implementation supports standard

TLS 1.3 cipher suites while maintaining the security guarantees of the MPC engine.

To implement AES-GCM-128 within our MPC framework, we use optimizations discussed in prior works [1, 28, 36]. We use the GC-optimized AES circuit from prior work [28], that minimizes the number of AND gates, taking advantage of state-of-the-art garbling protocols where XOR is a free operation. For GCM, we use the following tag-generation procedure. After deriving the TLS application key inside MPC, the parties jointly compute the GCM generator $H = E_K(0)$ along with its successive powers $H, H^2, \ldots, H^L$, which are secret-shared across all servers. To authenticate data blocks $S_1, S_2, \ldots, S_L$, each party locally evaluates its share of the polynomial $\sum_{i=1}^{L} S_i \cdot H^i$, and these shares are then combined with the MPC-encrypted initialization vector to produce the final GCM authentication tag.

We use the optimization in MPCAuth [28] to create 16 elements of the power series at a time, which allows us to compute the tag for 224 bytes. Thus, in our implementation we split up each request into 224 byte blocks before sending them to the server.

### 4.2 Creating a Phantom Identity

When the users invoke **CreateAccount**, the secure agent first creates a fresh *phantom identity*, which we explain below.

Cloud providers require that an account be tied to personal information (name, email, billing, etc.). Using one user's real identity for the secure agent's cloud account is dangerous, because that user can potentially use that information to take control of the secure agent's cloud account. For example, a user who knows and controls the name, email address, and/or billing information associated with the secure agent's cloud account could call customer support for the cloud provider claiming that they are the true owner of the account, and ask to reset the password. If they succeed in doing this, then they can log in to the secure agent's cloud account using the newly reset password, and perform actions bypassing the secure agent (e.g., to steal any data that the other users have shared with the secure agent's cloud account).

Flexroom avoids this by having the participants jointly create a *phantom identity*, a fake online persona that no single user controls. In Flexroom, the phantom identity includes a name, an email account (i.e., email address and password), and an untraceable payment method. These attributes are created within the MPC environment and stored in the secure agent's private state, ensuring that they can only be accessed within the secure computation and not by any individual user.

The name is chosen randomly by the parties with MPC. The email address and email password must be associated with a real email account, because signing up for a cloud account with an email address requires receiving an email from the cloud provider and reading it. Therefore, generating a phantom identity requires issuing API calls from MPC to an email provider (`mail.tm` in our implementation) to create an email account, whose email address and password are not revealed to Alice and Bob.

The above techniques hide the name and email account from Alice and Bob. For billing information, our implementation takes the simple approach of having one of the users simply input their billing information (e.g., card number) to the secure agent. Although the card number is known to the users, we believe that knowing the payment information, without name or email address on the account, is not enough to be able to call customer support and convince them to reset the password.

Still, it may be considered "more secure" to hide the billing information from the users too, so we explore how to do this (though we do not implement our technique below). The challenge

is that is the phantom identity's billing information cannot be hidden using the above techniques. While the secure agent can create a fresh email account as described above, it cannot open a credit card or bank account under a fake persona, due to the Know-Your-Customer (KYC) regulations. To work around this requirement, our insight is that AWS accepts *pre-paid debit cards* (e.g., VISA gift cards) as a form of payment, and that the process of purchasing such debit cards de-links the payment method from the real-world identity making the purchase. Our proposed approach is to have the secure agent purchase the pre-paid gift card. For example, the secure agent can (1) accept one of the user's billing information as input, (2) issue API calls over HTTPs to purchase a VISA "eGift" card online using the user's billing information, and (3) obtain the debit card number by downloading and parsing the PDF containing payment details inside MPC. At the end of this procedure, the payment information is stored in the secure agent's private state, but neither Alice nor Bob learns the payment information.

Using phantom identities, Flexroom creates a cloud account for a fictitious person, backed by a legitimate email account and payment information. These details satisfy AWS's sign-up requirements, but are not known to the users using Flexroom. From AWS's perspective, the provided information passes validation, so it is not obvious to AWS that the account truly corresponds to a phantom identity instead of an ordinary customer (with the caveat that timing and other side channels that Flexroom does not hide may give this away). Most importantly, users do not have enough information to claim ownership of the cloud account and reset its password via out-of-band, non-cryptographic, social channels (e.g., by calling customer support).

### 4.3 Authentication to AWS

For **CreateAccount** (after first creating a phantom identity) and **CreateSession**, the secure agent interacts with the AWS API endpoint. We explain that process in this section.

To register and authenticate an AWS account (in **CreateAccount** and **CreateSession**), Flexroom reproduces the entire AWS sign-up and login sequence via MPC. This involves issuing the same HTTPS requests that a normal user's web client would issue as they create and account and sign in using AWS's web interface. In other words, it issues the HTTPS requests for navigating the AWS registration pages, filling out forms with the phantom identity's details, handling verification challenges, and establishing credentials. These HTTPS requests are sent via the secure agent, with all sensitive cookies secured using MPC and TLS. We relegate a full discussion of the specific API calls and their challenges to §5.

In our secret-shared authentication protocol, cookies and user credentials are incrementally revealed to ensure that no single party can unilaterally complete a login. Specifically, Alice and Bob each hold complementary shares of the user's email address, which are combined only when issuing an authentication request. During the AWS login flow, the server emits several cookies: some non-essential, others security-critical (e.g., Set-Cookie: aws-creds =). Nonessential cookies are revealed in plaintext where Alice takes care of it. To protect sensitive cookies, the protocol reconstructs the server response byte by byte and stops further secret sharing as soon as the Set-Cookie: aws-creds= prefix for example is detected. At that checkpoint, reconstruction of the cookie value is suspended. With a garbled circuit dedicated to detecting semicolons inside text, Alice and Bob jointly find the next corresponding semicolon and jumps directly to the end of the value segment, leaving those bytes remaining secret shared and never combined to reveal it in plaintext.

After suspension, both parties store their shares of the critical cookies in secret-shared form. When a subsequent request requires these cookies, Alice transmits to Bob the index range within

the request payload where the cookie value should appear. Bob at first has a zero byte string that is of the same length as the request. After he receives the index range from Alice, Bob then fills the corresponding positions with his share of the cookie value, completing reconstruction in collaboration while preserving the secret-sharing guarantee until the final combination.

### 4.4 Using the Cloud

**RunScript** allows Alice and Bob to run a collaborative computation using the secure agent's cloud account (after they have previously run **CreateAccount** and **RunScript**). It issues HTTPS requests, similarly to what was described in §4.3. However, unlike the requests in §4.3, which are meant to be made from a user's web browser, these requests use AWS's cloud API that is meant to be used by headless computer programs. Thus, the API calls for **RunScript** tend to be simpler and more convenient to use.

One challenge is that AWS's API is large and constantly evolving; as a result, it may seem difficult to enable Flexroom to use all aspects of AWS's API, since each API call would need to be supported in MPC (e.g., reduced to string substitutions, as described in §5). Our insight is that we can leverage the cloud itself to avoid having to implement the entirety of AWS's ever-evolving API surface in MPC. Specifically, the secure agent needs to issue API calls to launch a virtual machine (EC2 instance), but can then install the cloud account credentials in the VM and have the VM issue additional API calls. The code running on the VM *does not need to issue API calls using MPC*, and can use AWS' own libraries (e.g., `boto3`) to allocate and use cloud services. Thus, to support **RunScript**, Flexroom only needs to use MPC-based secure agents to issue a small core of cloud API actions—to spawn a VM that executes the specified script. This can be accomplished using the standard AWS APIs, using `cloud-init` to specify the script that the VM should run once it boots.

For large-scale collaborations that benefit from clean rooms (e.g., collaborative data analytics [3, 29]), the collaborating users may wish to provide large datasets as input to the computation. A naïve strawman approach would be for the users to provide their datasets as input to the secure agent, which would upload the datasets to its cloud account by issuing API calls from MPC. The downside to this strawman approach is performance; because the entire dataset must be processed and encrypted (according to TLS) within MPC, uploading a large dataset in this way is likely to be very slow. Our insight is to have the users provide input by uploading their datasets directly to the cloud, in a way that is accessible to the secure agent but not to any other parties. Although the users do not know the identity of the secure agent's cloud account, they can share their data with it by creating pre-signed URLs, and providing those URLs as input to the secure agent. Our implementation has the users upload their datasets to Google Drive and share a link with the secure agent. We use this approach due to the familiarity and convenience of Google Drive's APIs; a downside is that it adds Google as an extra trusted party. This could easily be solved by having users upload their data to Amazon S3 (e.g., in their own AWS accounts) instead and using pre-signed URLs as described above. Once the datasets are shared with the secure agents in this way, the workload in the script passed to the **RunScript** can access the parties' input datasets directly from cloud storage.

| Request Name | Purpose | Inputs | Outputs | MPC |
|---|---|---|---|---|
| CreateEmail | Creates a temporary email | (Email) | None | Yes |
| InitialRequest | Get session token for signup | None | None | No |
| SubmitEmail | Send email for verification | (Email) | None | Yes |
| RequestCaptcha | Request the captcha for verfication | None | None | No |
| GetCaptchaImage | Get the image for the captcha | None | None | No |
| VerifyCaptcha | Sends the guess for the captcha | None | None | No |
| GetAccessCode | Get session code for signup | None | None | No |
| SubmitAccessCode | Gets new session token for Signup | (Email) | None | Yes |
| GetOTP | Get the OTP from your email | (Email) | None | Yes |
| VerifyOTP | Verifies the OTP from the email | (Email) | None | Yes |
| CompleteSignup | Complete account creation | (Email, Password) | (AWSCreds) | Yes |
| InitiateBilling | Initiate setting up billing | (AWSCreds) | None | Yes |
| SubmitInfo | Submits account information | (AWSCreds) | None | Yes |
| PrereqCredit | Submits prerequisite Credit Card Information | (AWSCreds) | None | Yes |
| SubmitCredit | Submits all Credit Card Information | (AWSCreds) | None | Yes |
| VerifyPhone | Verifies the account phone number | (AWSCreds) | None | Yes |

Table 1: List of requests for creating an AWS account in **CreateAccount**

# 5 Issuing API Calls

## 5.1 AWS Signup Endpoint

We started by looking at the requests that are needed to create an account when going through the AWS webpage. This consists of 16 requests in the **CreateAccount** process that can be seen in Table 1, along with what inputs and outputs from the requests need to be secret shared. All requests are made to Amazon's endpoints, with the exception of *CreateEmail* and *GetOTP* which are implemented using *mail.tm's* API. We can note that *CreateEmail* can be preprocessed before making a connection to Amazon for the rest of **CreateAccount**.

Each request is handled differently based on what data is required for input, as well as what is output in its response. For example, our initial *CreateEmail* request includes the address of the shared email, and thus we must construct the request with our TLS-in-MPC client. We combine them similar to the cookie process in the previous section.

We can note that this request does not return any important information, and thus, the response can be fully reconstructed without any loss of security. In contrast, the *CompleteSignup* request contains the cookie *AWSCreds* in its response, which is used in the requests to set up the information for the rest of the account. Because of the sensitivity of this cookie, we can not fully reconstruct the response from each party's shares like before, and instead must parse out the secret shares of the cookie from the secret shares of the response. We use the process in §4.3 to retrieve this cookie

Any verification steps required during account setup (for example, solving a CAPTCHA challenge or retrieving a confirmation code sent to the email) are handled collectively. One user might perform the human step (e.g., solving a CAPTCHA or entering an OTP code), but this yields no lasting advantage since the underlying credentials remain secret-shared. In this way, no one can reconstruct the phantom's secrets (email login, cloud password, etc.) on their own.

An interesting feature of the AWS API is that the password sent in the *CompleteSignup* request

| Request Name | Purpose | Inputs | Outputs | MPC |
|---|---|---|---|---|
| GetSigninConsole | Get session cookies for signin | None | None | No |
| GetSignin | Get session cookies for signin | None | None | No |
| SendEmail | Sends the email for the account | (Email) | None | Yes |
| SendCaptcha | Sends the guess for the Captcha | (Email) | None | Yes |
| CheckMFA | Checks if the account has MFA | (Email) | None | Yes |
| ConsoleSignin | Sends username and password to account | (Email, Password) | (AWSCreds) | Yes |
| ChallengeOauth | Complete the Oauth code challenge | (AWSCreds) | None | Yes |
| ConsoleRedirect1 | Redirects and resets cookies | (AWSCreds) | (AWSCreds) | Yes |
| ConsoleRedirect2 | Redirects and resets cookies | (AWSCreds) | None | Yes |
| CreateAWSCredentials | Creates AWS Credentials for the account | (AWSCreds) | (Creds.AccessKey, Creds.SecretKey) | Yes |

Table 2: List of requests for signing into AWS in **CreateSession**

is not the plaintext of the password, but rather a base64 encoding of a JSON Web Encryption (JWE) which contains the password when decrypted. This consists of first encrypting the password with AES-GCM and a random key, and then encrypting the key with RSA-OAEP using a public key found in the meta tags of the AWS website. This object is then strung together before being sent in the request. Because the password is sensitive information, it is important that both the encryption of the password with RSA and the key with AES-GCM happen in MPC. We discuss this further in §6.2.

### 5.2 AWS Signin Endpoint

We also inspected the AWS Sign-in API process that is performed when signing into AWS through the browser, allowing us to simulate a browser sign-in using our TLS-in-MPC client. We found there to be ten requests crucial to the **CreateAccount** process, listed in Table 2.

After signing into the console, a cookie is set called *AWSCreds* that enables our client to perform operations in the console using our account, similar to in **CreateAccount**. Due to the sensitivity of this cookie, it must be secret-shared between the two parties.

An interesting quirk in the sign-in process is the request parameter called *metadata1*. This field normally consists of a JSON object containing data about the system and browser used to sign in through the AWS website. The JSON object is then encrypted by the XXTEA cipher with a constant key before being sent in each request [18]. We noticed that simply replaying *metadata1* causes requests to fail, possibly indicating a type of anti-spam detection. In order to bypass this detection, we first create our own metadata JSON with randomized noise. Afterwards, we can encrypt it with the XXTEA key that can be found in the sign-in page.

Once the **CreateAccount** process is completed, a JSON object that we call *Creds* is output. This JSON object contains the AWS access key as well as the AWS secret key that are used to make API calls to the account for **RunScript**, which we refer to as *Creds.AccessKey* and *Creds.SecretKey*. Since Creds.SecretKey can be used to pilot the AWS account, it is important that its value remains secret shared between both parties. Although parsing a JSON in MPC is hard, we can simplify the process as we already know the format of Creds. Thus, we can simply index the JSON string for the shares of Creds.SecretKey without having to parse it.

### 5.3 AWS SDK Endpoint

In order to spawn an EC2 instance for **RunScript**, we replicated the same requests as the AWS SDK *Boto3* through our TLS-in-MPC framework. This consists of ten total requests we ran to spawn an EC2 instance and run a Sagemaker job, which can be seen in Table 3.

| Request Name | Purpose | Inputs | Outputs | MPC |
|---|---|---|---|---|
| CreateIamRole | Create an IAM role for Sagemaker | (Creds.SecretKey) | None | Yes |
| AttachSagePolicy | Attach a policy to the new role | (Creds.SecretKey) | None | Yes |
| AttachS3Policy | Attach a policy to the new role | (Creds.SecretKey) | None | Yes |
| CreateS3Bucket | Create an S3 Bucket | (Creds.SecretKey) | None | Yes |
| GetAmiID | Get newest Linux AMI Id | (Creds.SecretKey) | None | Yes |
| CheckKeyPair | Check to see if a key pair exists | (Creds.SecretKey) | None | Yes |
| CreateKeyPair | Create a key pair if it does not exist | (Creds.SecretKey) | None | Yes |
| CreateSecurityGroup | Create a security group for the EC2 instance | (Creds.SecretKey) | None | Yes |
| StartEC2Instance | Start an EC2 instance | (Creds.SecretKey) | None | Yes |
| CreateSagemakerJob | Create the Sagemaker training Job | (Creds.SecretKey) | None | Yes |

Table 3: List of requests for spawning an EC2 Instance in **RunScript**

The unique challenge of interacting with the AWS SDK endpoint is implementing the *AWS SigV4* protocol that is used to authenticate each request. This process consists of iteratively computing the HMAC of the date, region, service, and finally the request. Each HMAC uses the previous HMAC as the key for the next, except for the first, which uses the secret key derived from signin. We can write the process as

$$\text{SigDate} = \text{HMAC}(\text{Creds.SecretKey}, \text{date}) \tag{1}$$

$$\text{SigRegion} = \text{HMAC}(\text{SigDate}, \text{region}) \tag{2}$$

$$\text{SigService} = \text{HMAC}(\text{SigRegion}, \text{service}) \tag{3}$$

$$\text{Signature} = \text{HMAC}(\text{SigService}, \text{request}) \tag{4}$$

In order to preserve the security guarantee of Flexroom, we perform each HMAC request in MPC where each party contributes their share of the secret key input and then receives their share of the signature output. This ensures that they will not be able to make each request independently.

## 6 Implementation

We developed a proof-of-concept prototype for Flexroom and secure agents. This includes an implementation of a TLS endpoint in MPC, as described in §4.1. As noted in §4.1, our protocol

builds on prior work, but no prior implementation of a TLS client in MPC is available for us to use in our implementation. Therefore, we implemented a TLS client in MPC ourselves. This required combining multiple general-purpose MPC frameworks in a multi-process execution model, which we describe below.

## 6.1 MPC Frameworks

Our implementation combines two state-of-the-art MPC frameworks: MP-SPDZ [14] for computations over arithmetic domains (specifically elliptic curve groups) and the EMP Toolkit [32] for computations over the binary domain. Our implementation is run through a Python driver program that handles each network request and invokes each program as a subprocess, as the two frameworks are separate processes.

### 6.1.1 MP-SPDZ

Leveraging existing MP-SPDZ support for SPDZ-style authenticated shares over elliptic curve groups [27], we implement the Diffie–Hellman protocol [1] within the MPC setting. We also provide the first implementation of daPoint [1], which enables efficient conversion from authenticated shares of elliptic curve points to authenticated shares of affine coordinates over finite fields. In addition, we use MP-SPDZ's built-in support for edaBits [9] to convert shares over finite fields into authenticated Boolean shares. These edaBits are preprocessed in advance, ensuring that the online phase—which executes after the TLS connection is established—remains lightweight and efficient.

### 6.1.2 EMP Toolkit

We use EMP-Toolkit's [32] `emp-agmpc` library, which provides an authenticated garbling protocol, to implement the remainder of the TLS protocol—including key derivation and the record layer—over the binary domain. This stage consumes the authenticated Boolean shares produced in the arithmetic phase. For subcircuits such as key derivation, AES, and the generation of $H$-series shares, we rely on the Bristol-format circuits provided by [28], which are open-sourced as standalone components (though not as a full end-to-end implementation). We further make use of their extension to EMP-Toolkit that supports reactive functionalities, allowing authenticated secret shares to serve directly as inputs to the garbling protocol. This enables us to compose our MPC program modularly by chaining circuits together rather than constructing monolithic circuits for each step, thereby improving clarity and maintainability. Finally, we extend this "chained execution" mechanism to support preprocessing across multiple calls, yielding a significantly faster online phase. This optimization introduces a memory-latency tradeoff but is well-suited to our application: while preprocessing may be computationally heavy, it is performed prior to connection establishment, ensuring that the online phase completes within the required timeout constraints.

## 6.2 Limitations of our Prototype

When the secure agent specifies the AWS account password in **CreateAccount**, Amazon's protocol has the following complication: The client must encrypt the user's password under AES-GCM with a random key, then encrypt the key under a certain RSA public key, and then transfer the resulting ciphertext to Amazon as part of the HTTPS request. In principle, the secure agent can achieve this by randomly choosing a password (§4.2), encrypting it with AES-GCM in MPC, and then

encrypting the random key under the public key within MPC. This encryption can happen offline (i.e., before establishing a connection with Amazon's web service) so it is not subject to any AWS timeout; we also believe that the RSA encryption can be made efficient using multiplicative secret sharing. Since pre-processing is not part of the critical workload timeline, we have omitted this pre-processing feature of implementation in our proof of concept prototype, and simply have one of the parties input the password encrypted under the AWS RSA public key. In a full-fledged implementation, this has to be replaced with secure password generation in the pre-processing step since AWS Public Key is known a priori.

For simplicity, our implementation has one party input billing information to the secure agent. As discussed in §4.2, an alternative approach, which hides the billing information from both users, is to have the secure agent issue API calls to purchase a VISA eGift Card.

A cloud account is subject to *service quotas*, which are limits on resource usage (e.g., on the number of VM CPUs that can be allocated for a given region and a VM type). A cloud tenant can request to raise their limit by filing a request online; such requests are approved subject to resource demand and capacity. In principle, Flexroom can support this by having the secure agent issue API calls to file such requests upon agreement by the users. In cases where contact with customer support is required, the secure agent can read its email account and output any emails received from customer service (with the appropriate redactions, as needed), and can ensure that all parties agree on any email responses sent as part of the process. For simplicity, however, our implementation does not support requesting to raise the service quotas. Furthermore, approval of certain limit increases may be contingent on further validation of the cloud user's information or use of certain billing methods (e.g., invoice billing). Thus, our phantom-identity-based approach, using a pre-paid debit card, may be ineligible for certain service quota increases.

Our implementation has the users provide input to MPC using Google Drive. As discussed in §4.4, this is not fundamental, and we could have used Amazon S3 to avoid including Google as an additional trusted party.

Our prototype handles the secure agent's private state as explicit inputs and outputs of each Flexroom operation, instead of by automatically writing to and reading from persistent storage. This means that Alice or Bob would have to manually collect their share of the secure agent's private state from the output of one step and input that share to the next step (e.g. when transitioning from **CreateAccount** to **CreateSession** or **CreateSession** to **RunScript**).

Finally, we note that we observed minor changes in the APIs used in **CreateAccount** and **CreateSession** (Table 1 and Table 2) over the course of developing our implementation. In our experience, the APIs change slightly every few weeks. This would not affect a normal user using a web browser, as the client code running in the web browser would be updated according to the new APIs. However, it does mean that Flexroom must be regularly maintained with ongoing changes to keep up with the latest APIs.

# 7 Evaluation

## 7.1 Comparison to Existing Clean Rooms

Table 4 lists limitations of some other existing clean room products, for comparison to Flexroom.

In general, clean rooms offered by the cloud provider are limited to particular kinds of computation (e.g., SQL). In contrast, Flexroom allows for arbitrary computations.

Third-party clean rooms, such as Databricks's clean room product, are sometimes more general, allowing for executing notebooks. However, Flexroom provides for qualitatively better security and functionality as follows:

- Such third-party clean rooms require trust in both the third party (Databricks), and the cloud provider where the code executes (AWS). In this respect, third-party clean rooms are like the CRSP strawman in §1 (where the third-party is the CRSP), and Flexroom improves upon them by eliminating the need to trust the CRSP.
- Flexroom supports computation that make use of cloud services (e.g., PaaS and SaaS products), which are not natively available even in third-party clean rooms.
- Looking ahead, secure agents can support both threshold MPC [4, 11], and general access structure MPC [13] with MPC participants deciding just among themselves what kind of MPC (with state) to run (see §8). In contrast, currently, Databricks only supports the unanimous agreement of all participants to allow the execution of specified notebook code, and any modification or extension to this rule must be implemented as a new Databricks functionality, currently not supported. In contrast, our secure agents can support any such permission structure without even having to disclose it to the cloud provider.

## 7.2 Performance of Flexroom

### 7.2.1 Experimental Methodology

We evaluate our Flexroom prototype by placing Alice's and Bob's compute resources in Microsoft Azure, and using Flexroom to orchestrate a computation in Amazon Web Services (AWS). Alice and Bob each run on a `Standard_E8bs_v5` instance, and they are placed in two different but nearby regions, namely East US and East US 2. These regions are both in Virginia, and therefore network performance between them is relatively good. Using `ping`, we observed a round-trip time of 6.6 ms and using `iperf3`, we observed TCP bandwidth of 5.55 Gbit/s. They are, however, distinct cloud regions of Microsoft Azure, providing a degree of separation between Alice and Bob.

### 7.2.2 Setup Performance

In this section, we characterize the performance of Flexroom's **CreateSession** operation. This sets up a session for collaborative computation, which can be used repeatedly. We did not measure **CreateAccount** in the same experimental setup (due to what appears to be a recent change in Amazon's APIs; see §6.2), but our experience is that its performance is similar to **CreateSession**.

In our evaluation setup, **CreateSession** takes 2725.4 seconds to run. We consider this acceptable as these operations constitute a *setup* phase for Flexroom that does not scale with the size of the computation. The session with AWS that results from the above operations can be used for a long collaborative computation or for multiple back-to-back collaborative computations.

We also characterize the performance of these operations by presenting performance breakdowns (Table 5). The "Handshake" and "Sending requests" components represent the computation that takes place while connected to the endpoint, and which are subject to the AWS timeout. Most of the total time for each request is not attributable to these components, indicating that the runtime is dominated by computation that occurs *between* the network connections: performing

| AWS Clean Rooms Limitations | Databricks Clean Rooms Limitations |
|---|---|
| **Restricted governance.** Clean Rooms allows partners to run queries checked against restricted "analysis rules", not general queries that participants agree on. See-AWS-Clean-Rooms | **Restricted Governance.** Databricks Clean Rooms use Delta Sharing. It allows view of certain columns/rows in a very restricted setting. See-Databricks—What-is-Clean-Rooms? |
| **Limited Programmability.** Can only run `SELECT` SQL (within allowed constructs), not arbitrary code. See-AWS-Analysis-Rules | **Limited Programmability.** Can only execute approved notebook code, not arbitrary computation involving cloud services. See-Databricks—Run-notebooks |
| **Blocking Select Reveal Commands.** Select Reveal on JOIN computations is not allowed. This precludes applications such as secure PSI (Private Set Intersection), where certain data is selectively revealed. See-C3R—Column-Types | **Blocking Select Reveal Commands.** It is not allowed for participants to selectively see each other's raw data, such as in PSI applications. See-Databricks |
| **No computation on "sealed" columns.** The so-called "sealed" columns allow SELECT, but do not allow general computation on their entries. See-AWS-C3R—Supported-Types | **No computation on columns of the table.** Databricks does not allow per-column arbitrary computation on the tables, though it does allow calls to TEEs. See-Azure-Confidential-Computing |
| **Restricted Differential Privacy (DP) use.** DP can only be used inside SQL structured data in Amazon S3, not general-purpose DP computation. See-AWS-DP | **No DP use whatsoever.** See-Databricks |
| **Limited add-on participation.** It is not allowed to add new members after clean-room creation. See-AWS-Collaboration-For-Queries | **Small size of allowed collaborators.** A clean room can include at most ten collaborators, including the creator of the clean room. See-Databricks |

Table 4: Comparison of secure agent general-purpose computation with AWS Clean Rooms and Databricks.

the "preprocessing" stage of MPC to prepare for each request, and parsing and processing the HTTP and JSON responses from the AWS web endpoint in MPC as well as decrypting the server response. This parsing is necessary to securely extract cookies and other sensitive information in order to complete the authentication workflows. This suggests that our techniques in §5 for efficiently parsing responses from the server in MPC are important for achieving good performance, and further improvements to this step could lead to meaningfully better performance.

The total memory usage of each client fits within the available 64 GiB at each node. Most of the memory consumption comes from storing preprocessed material from the function-independent and function-dependent "offline" phases of the MPC protocols. While these phases can sometimes be performed "on the fly" with the online phase in order to save memory, our implementation performs these phases at the beginning of each request, before establishing a network connection to the AWS endpoint; we do this so that only the online phase needs to be performed after establishing a connection, so that we can avoid request-level timeouts (§5). The secure agent's private state, which is populated by the **CreateAccount** and **CreateSession** operations, is secret-shared between Alice and Bob. This private state requires less than a kilobyte of persistent storage for each of Alice and Bob.

| Request Name | Number of 224 bytes blocks | Handshake | Sending requests | Update State | Total time |
|---|---|---|---|---|---|
| SendEmail | 48 | 1.39 | 31.66 | NA | 132.45 |
| SendCaptcha | 58 | 1.39 | 38.46 | NA | 151.36 |
| CheckMFA | 11 | 1.38 | 7.56 | NA | 40.35 |
| ConsoleSignin | 52 | 1.37 | 35.46 | 627.87 | 775.12 |
| ChallengeOauth | 22 | 1.36 | 14.79 | NA | 68.53 |
| ConsoleRedirect1 | 29 | 1.37 | 19.09 | 206.55 | 289.64 |
| ConsoleRedirect2 | 22 | 1.36 | 15.17 | NA | 1198.81 |
| CreateAWSCredentials | 24 | 1.38 | 16.13 | NA | 67.92 |

Table 5: Breakdown of the MPC steps for **CreateSession** in seconds

| Request Name | Number of 224 bytes blocks | Handshake | Sending requests | AWS SigV4 | Total time |
|---|---|---|---|---|---|
| CreateIamRole | 4 | 1.39 | 2.69 | 2.73 | 23.88 |
| AttachSagePolicy | 3 | 1.37 | 2.02 | 2.71 | 20.87 |
| AttachS3Policy | 3 | 1.38 | 2.08 | 2.71 | 21.03 |
| CreateS3Bucket | 2 | 1.36 | 1.43 | 2.69 | 18.01 |
| GetAmiID | 3 | 1.38 | 2.01 | 2.71 | 20.47 |
| CheckKeyPair | 3 | 1.39 | 2.00 | 2.72 | 21.50 |
| CreateKeyPair | 3 | 1.39 | 2.10 | 2.72 | 20.92 |
| CreateSecurityGroup | 3 | 1.38 | 2.06 | 2.70 | 23.19 |
| StartEC2Instance | 89 | 1.38 | 60.85 | 2.73 | 237.32 |
| CreateSagemakerJob | 7 | 1.36 | 4.71 | 2.71 | 30.62 |

Table 6: Breakdown of the MPC steps for **RunScript** in seconds

#### 7.2.3 Collaborative Computation Performance

Now, we characterize the performance of Flexroom's **RunSession** operation to run a collaborative computation on behalf of Alice and Bob. For our evaluation, we use **RunSession** to run a machine learning workload using Amazon SageMaker, where Alice brings an input dataset and Bob brings a training script. Note that this workload could not be run using Amazon's existing clean room product, because it is limited in functionality and does not include PaaS offerings such as Amazon SageMaker. The fact that we can run this workload at all shows the flexibility afforded by Flexroom's design.

The results are shown in Table 6. The full process of invoking **RunSession** takes about 7.3 minutes, not including running the workload. Note that this is a constant, additive overhead, no matter what computation is run. Even if the collaborative workload takes several hours to run, running it using Flexroom would slow it down by the same amount.

### 7.3 Opportunities for Optimization

There are multiple opportunities to optimize the performance of Flexroom. We focus on the **RunSession** operation; the others, **CreateAccount** and **CreateSession**, can be considered "pre-

processing" in that they can be completed before a collaborative computation actually needs to take place.

Our implementation does not fully use the technique in §4.4. Specifically, the "CreateSageMakerJob" API call in Table 6 is issued directly by the secure agent, from MPC. To improve performance, we can fully utilize that technique, to issue this API call in plaintext from the VM, instead of from MPC.

Taking this further, we believe that we can remove more of the API calls in Table 6 from MPC, either by eliminating them altogether or by issuing them from the VM in plaintext. For example, API calls like "AttachSagePolicy" can be issued from the spawned VM. API calls like "GetAmiID" can be issued independently by Alice and Bob, and the AMI ID then included directly in the request to issue. It may be possible to re-implement **RunScript** to issue a single API call, "StartEC2Instance," from MPC, with the rest either eliminated or issued quickly in plaintext. We believe that "StartEC2Instance" may be slow because it is large (89 blocks, as shown in Table 6). To make "StartEC2Instance" smaller, we can use a design similar to bootloaders in operating systems. Specifically, the `cloud-init` script to execute, included in the "StartEC2Instance" API call, can be made into a small bootloader-like script that fetches the "real" script from Amazon S3 and executes it. Importantly, Alice and Bob *both* upload the desired code to their Amazon S3 accounts; the bootloader-like script in "StartEC2Instance" would include the URLs of both scripts, download them both, check that they are equal, and execute one of the scripts only if they are equal. This procedure guarantees that both Alice and Bob agree on the computation to run, while reducing the size (and therefore the performance) of the "StartEC2Instance" API call.

# 8 Future Work, Discussion, and Conclusion

In this paper, we presented Flexroom, a new clean room design that allows users wishing to collaborate to use any available service in their chosen cloud provider as part of their collaborative computation. Flexroom is based on a new system design element called a *secure agent*, which works by running a TLS client within a maliciously-secure, reactive MPC computation. This design allows the two parties to issue API calls to the cloud provider, while keeping the cloud credentials hidden from the two parties and enforcing unanimous agreement for each API call. We implemented Flexroom for the case of $n = 2$ users using Amazon Web Services (AWS) for their collaborative computation, showing that our design ideas are practical.

Flexroom needs to use phantom identities and run a TLS client inside MPC, because we designed Flexroom to work with existing cloud providers like AWS, without requiring any changes to cloud infrastructure. A clean slate setting, in which changes to cloud infrastructure are desirable, may lead to a fundamentally more efficient design for Flexroom. For example, if the cloud provider is *aware* of and *directly supports* the use case where a cloud account is jointly controlled by multiple users, then it can allow them to control it directly, without MPC and without a phantom identity. Concretely, the cloud provider could accept requests from all $n$ users, each of whom authenticate separately, and only perform the requested action if all of the $n$ users' requests agree—in effect, the cloud provider takes responsibility for enforcing unanimous agreement. It could also split the cost for the account among the users, and enact policies that require *all* users on the account to agree to reset the password.

The above idea requires the cloud provider to be aware of which cloud accounts are being used for collaborative computation. An alternative research direction is to keep Flexroom's design based

on secure agents, and instead make it more difficult for the cloud provider to identify which cloud accounts are controlled by secure agents. Flexroom's current design does not hide side channels or timing; with future research, it may be possible to obscure side channels, so that a cloud provider cannot use them to identify cloud accounts controlled by secure agents. This may improve security, as an attacker who is an employee of the cloud provider may find it more difficult to identify collaborative computations with sensitive data to stealthily attack.

In designing Flexroom, we manually converted each API call to make within MPC into a set of string substitution rules, and developed optimizations for parsing HTTP and JSON responses inside MPC to extract sensitive information (e.g., cookies) that we manually identified (§5). In Flexroom's case, it is not necessary to issue general API calls, because we can use a VM spawned in the cloud to issue additional API calls. This works because we are assuming that AWS is a trusted party. However, users may with to use secure agents to collaborate using non-cloud services (e.g., OpenAI); in such cases spawning a VM to issue API calls is undesirable because it requires adding a cloud provider, like AWS, as an additional trusted party in the system. To support such use cases, an interesting research direction is to build a library that generalizes and exposes our techniques from §5.

Finally, secure agents can also be used with threshold-secure MPC algorithms [4, 11] or MPC algorithms that support access structures [13]. Such a secure agent could enforce a rich and flexible form of access control for deciding which API calls are allowed. For example, it can be programmed to take an action when a predefined threshold of the independent parties (could be a majority of parties, unanimous agreement, or anything in between, for example a required participation of specific subset of parties). The access policies could even require a specific action to occur periodically, have secret policies indicating when or how often an action is allowed or not allowed to be executed, or mandate post-processing of a query result (e.g., to add differential privacy).

## References


[1] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious TLS via multi-party computation. In *Topics in Cryptology – CT-RSA 2021*, pages 51–74, 2021.

[2] Amazon. Data collaboration service - AWS clean rooms - AWS. https://aws.amazon.com/clean-rooms/. Accessed: September 6, 2024.

[3] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: secure querying for federated databases. *Proc. VLDB Endow.*, 10(6):673—-684, 2017.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 1–10, 1988.

[5] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod $2^k$ for dishonest majority. Cryptology ePrint Archive, Paper 2018/482, 2018.

[7] Anders Dalskov, Marcel Keller, Claudio Orlandi, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. Cryptology ePrint Archive, Paper 2019/889, 2019.

[8] Erez Eizenman. Scotiabank's chief risk officer on the state of anti-money laundering. https://www.mckinsey.com/capabilities/risk-and-resilience/our-insights/scotiabanks-chief-risk-officer-on-the-state-of-anti-money-laundering, 2019. Accessed: September 5, 2024.

[9] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. Cryptology ePrint Archive, Paper 2020/338, 2020.

[10] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[11] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, 1987.

[12] Google. Secure and privacy-centric sharing with data clean rooms in BigQuery. https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-data-clean-rooms. Accessed: September 8, 2024.

[13] Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, April 2000.

[14] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Paper 2016/505, 2016.

[16] Sam Kumar. *Rethinking System Design for Expressive Cryptography*. PhD thesis, University of California, Berkeley, 2023.

[17] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: Nearly zero-cost virtual memory for secure computation. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 367–385. USENIX Association, 2021.

[18] mkb79. How can I properly mimic this encryption method to produce the proper value for the encryptedPwd field? https://stackoverflow.com/questions/67139708/how-can-i-properly-mimic-this-encryption-method-to-produce-the-proper-value-for. Accessed: September 2, 2025.

[19] National Academies. To increase pace and volume of drug approvals for rare diseases, report recommends FDA enhance information sharing, improve collaboration. Accessed: September 24, 2024.

[20] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*, pages 334–348, 2013.

[21] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously-Secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2129–2146. USENIX Association, August 2021.

[22] Lucy Qin. Deploying MPC for social good. Real World Crypto, 2019. https://youtu.be/5pkDq4sRWyQ?t=2090.

[23] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[24] James Reyes. Building the next generation of digital advertising with MPC. Real World Crypto, 2022. https://youtu.be/6Gb0xO8csVU?t=2533.

[25] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and boolean circuits with active security. Cryptology ePrint Archive, Paper 2019/207, 2019.

[26] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612—-613, 1979.

[27] Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. Cryptology ePrint Archive, Paper 2019/768, 2019.

[28] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. MPCAuth: Multi-factor authentication for distributed-trust systems. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 829–847, 2023.

[29] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[30] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2021(1):188–208, 2021.

[31] Amanda Walker, Sarvar Patel, and Moti Yung. Helping organizations do more without collecting more data. Google Security Blog, 2019. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html.

[32] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[33] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 39–56, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 827–844, 2022.

[35] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. Cryptology ePrint Archive, Paper 2019/1104, 2019.

[36] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919—-1938, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 724–738, 2019.